7 IP VERSION 4

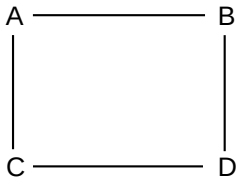
There are multiple LAN protocols below the IP layer and multiple transport protocols above, but IP itself stands alone. The Internet is essentially the IP Internet. If you want to run your own LAN protocol somewhere, or if you want to run your own transport protocol, the Internet backbone will still work just fine for you. But if you want to change the IP layer, you will encounter difficulty. (Just talk to the IPv6 people, or the IP-multicasting or IP-reservations groups.)

Currently the Internet uses (mostly, but no longer quite exclusively) IP version 4, with its 32-bit address size. As the Internet has run out of new large blocks of IPv4 addresses (*1.10 IP - Internet Protocol*), there is increasing pressure to convert to IPv6, with its 128-bit address size. Progress has been slow, however, and delaying tactics such as IPv4-address markets and NAT (*7.7 Network Address Translation*) – by which multiple hosts can share a single public IPv4 address – have allowed IPv4 to continue. Aside from the major change in address structure, there are relatively few differences in the routing models of IPv4 and IPv6. We will study IPv4 in this chapter and IPv6 in the following; at points where the IPv4/IPv6 difference doesn't much matter we will simply write "IP".

IPv4 (and IPv6) is, in effect, a universal **routing and addressing** protocol. Routing and addressing are developed together; every node has an IP address and every router knows how to handle IP addresses. IP was originally seen as a way to *inter*connect multiple LANs, but it may make more sense now to view IP as a virtual LAN overlaying all the physical LANs.

A crucial aspect of IP is its **scalability**. As the Internet has grown to ~10^9 hosts, the forwarding tables are not much larger than 10^5 (perhaps now $10^{5.5}$). Ethernet, in comparison, scales poorly.

Furthermore, IP, unlike Ethernet, offers excellent support for **multiple redundant links**. If the network below were an IP network, each node would communicate with each immediate neighbor via their shared direct link. If, on the other hand, this were an Ethernet network with the spanning-tree algorithm, then one of the four links would simply be disabled completely.



The IP network service model is to act like a giant LAN. That is, there are no acknowledgments; delivery is generally described as best-effort. This design choice is perhaps surprising, but it has also been quite fruitful.

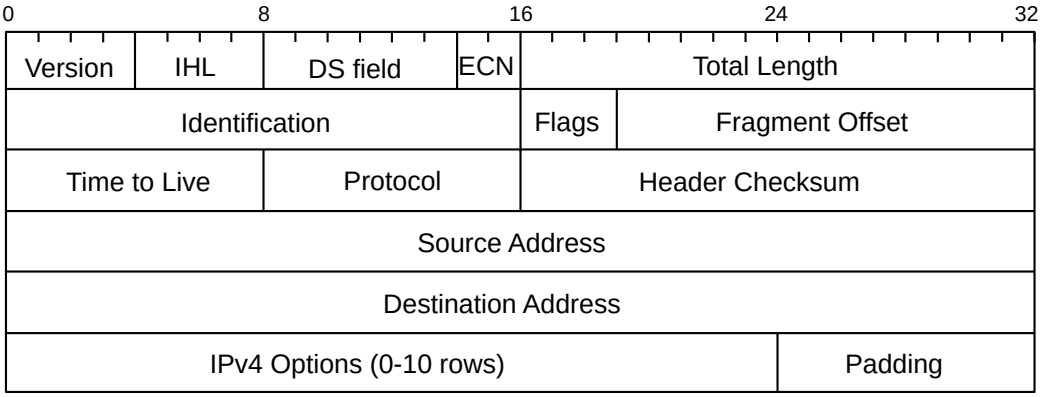
If you want to provide a universal service for delivering any packet anywhere, what else do you need besides routing and addressing? Every network (LAN) needs to be able to carry any packet. The protocols spell out the use of octets (bytes), so the only possible compatibility issue is that a packet is *too large* for a given network. IPv4 handles this by supporting **fragmentation**: a network may break a too-large packet up into units it can transport successfully. While IPv4 fragmentation is inefficient and clumsy, it does guarantee that any packet can potentially be delivered to any node. (Note, however, that IPv6 has given up on universal fragmentation; *8.5.4 IPv6 Fragment Header*.)

7.1 The IPv4 Header

The IPv4 Header needs to contain the following information:

- destination and source addresses
- indication of ipv4 versus ipv6
- a Time To Live (TTL) value, to prevent infinite routing loops
- a field indicating what comes next in the packet (*eg* TCP v UDP)
- fields supporting fragmentation and reassembly.

The header is organized as a series of 32-bit words as follows:



The IPv4 header, and basics of IPv4 protocol operation, were originally defined in **RFC 791**; some minor changes have since occurred. Most of these changes were documented in **RFC 1122**, though the DS field was defined in **RFC 2474** and the ECN bits were first proposed in **RFC 2481**.

The **Version** field is, for IPv4, the number 4: 0100. The **IHL** field represents the total IPv4 Header Length, in 32-bit words; an IPv4 header can thus be at most 15 words long. The base header takes up five words, so the IPv4 Options can consist of at most ten words. If one looks at IPv4 packets using a packet-capture tool that displays the packets in hex, the first byte will most often be 0x45.

The **Differentiated Services** (DS) field is used by the Differentiated Services suite to specify preferential handling for designated packets, *eg* those involved in VoIP or other real-time protocols. The **Explicit Congestion Notification** bits are there to allow routers experiencing congestion to mark packets, thus indicating to the sender that the transmission rate should be reduced. We will address these in *14.8.3 Explicit Congestion Notification (ECN)*. These two fields together replace the old 8-bit **Type of Service** field.

The **Total Length** field is present because an IPv4 packet may be smaller than the minimum LAN packet size (see Exercise 1) or larger than the maximum (if the IPv4 packet has been fragmented over several LAN packets. The IPv4 packet length, in other words, cannot be inferred from the LAN-level packet size. Because the Total Length field is 16 bits, the maximum IPv4 packet size is 2^{16} bytes. This is probably much too large, even if fragmentation were not something to be avoided (though see IPv6 "jumbograms" in *8.5.1 Hop-by-Hop Options Header*).

The second word of the header is devoted to fragmentation, discussed below.

The **Time-to-Live** (TTL) field is decremented by 1 at each router; if it reaches 0, the packet is discarded. A typical initial value is 64; it must be larger than the total number of hops in the path. In most cases, a value of 32 would work. The TTL field is there to prevent routing loops – always a serious problem should they occur – from consuming resources indefinitely. Later we will look at various IP routing-table update protocols and how they minimize the risk of routing loops; they do not, however, eliminate it. By comparison, Ethernet headers have no TTL field, but Ethernet also disallows cycles in the underlying topology.

The **Protocol** field contains a value to identify the contents of the packet body. A few of the more common values are

- 1: an ICMP packet, *7.11 Internet Control Message Protocol*
- 4: an encapsulated IPv4 packet, *7.13.1 IP-in-IP Encapsulation*
- 6: a TCP packet
- 17: a UDP packet
- 41: an encapsulated IPv6 packet, *8.13 IPv6 Connectivity via Tunneling*
- 50: an Encapsulating Security Payload, *22.11 IPsec*

A list of assigned protocol numbers is maintained by the IANA.

The **Header Checksum** field is the "Internet checksum" applied to the header only, not the body. Its only purpose is to allow the discarding of packets with corrupted headers. When the TTL value is decremented the router must update the header checksum. This can be done "algebraically" by adding a 1 in the correct place to compensate, but it is not hard simply to re-sum the 8 halfwords of the average header. The header checksum must also be updated when an IPv4 packet header is rewritten by a NAT router.

The **Source** and **Destination Address** fields contain, of course, the IPv4 addresses. These would normally be updated only by NAT firewalls.

The source-address field is supposed to be the sender's IPv4 address, but hardly any ISP checks that traffic they send out has a source address matching one of their customers, despite the call to do so in **RFC 2827**. As a result, IP **spoofing** – the sending of IP packets with a faked source address – is straightforward. For some examples, see *12.10.1 ISNs and spoofing*, and SYN flooding at *12.3 TCP Connection Establishment*.

IP-address spoofing also facilitates an all-too-common IP-layer **denial-of-service** attack in which a server is flooded with a huge volume of traffic so as to reduce the bandwidth available to legitimate traffic to a trickle. This flooding traffic typically originates from a large number of compromised machines. Without spoofing, even a lengthy list of sources can be blocked, but, with spoofing, this becomes quite difficult.

One IPv4 option is the **Record Route** option, in which routers are to insert their own IPv4 address into the IPv4 header option area. Unfortunately, with only ten words available, there is not enough space to record most longer routes (but see *7.11.1 Traceroute and Time Exceeded*, below). The **Timestamp** option is related; intermediate routers are requested to mark packets with their address and a local timestamp (to save space, the option can request only timestamps). There is room for only four ⟨address,timestamp⟩ pairs, but addresses can be **prespecified**; that is, the sender can include up to four IPv4 addresses and only those routers will fill in a timestamp.

Another option, now deprecated as security risk, is to support **source routing**. The sender would insert into the IPv4 header option area a list of IPv4 addresses; the packet would be routed to pass through each of those IPv4 addresses in turn. With **strict** source routing, the IPv4 addresses had to represent adjacent neighbors; no router could be used if its IPv4 address were not on the list. With **loose** source routing, the

listed addresses did not have to represent adjacent neighbors and ordinary IPv4 routing was used to get from one listed IPv4 address to the next. Both forms are essentially never used, again for security reasons: if a packet has been source-routed, it may have been routed outside of the at-least-somewhat trusted zone of the Internet backbone.

Finally, the IPv4 header was carefully laid out with memory alignment in mind. The 4-byte address fields are aligned on 4-byte boundaries, and the 2-byte fields are aligned on 2-byte boundaries. All this was once considered important enough that incoming packets were stored following two bytes of padding at the head of their containing buffer, so the IPv4 header, starting after the 14-byte Ethernet header, would be aligned on a 4-byte boundary. Today, however, the architectures for which this sort of alignment mattered have mostly faded away; alignment is a non-issue for ARM and Intel x86 processors.

7.2 Interfaces

IP addresses (both IPv4 and IPv6) are, strictly speaking, assigned not to hosts or nodes, but to **interfaces**. In the most common case, where each node has a single LAN interface, this is a distinction without a difference. In a room full of workstations each with a single Ethernet interface `eth0` (or perhaps `Ethernet adapter Local Area Connection`), we might as well view the IP address assigned to the interface as assigned to the workstation itself.

Each of those workstations, however, likely also has a **loopback** interface (at least conceptually), providing a way to deliver IP packets to other processes on the same machine. On many systems, the name "localhost" resolves to the IPv4 loopback address 127.0.0.1 (the IPv6 address ::1 is also used); see *7.3 Special Addresses*. Delivering packets to the loopback interface is simply a form of interprocess communication; a functionally similar alternative is named pipes.

Loopback delivery avoids the need to use the LAN at all, or even the need to *have* a LAN. For simple client/server testing, it is often convenient to have both client and server on the same machine, in which case the loopback interface is a convenient (and fast) standin for a "real" network interface. On unix-based machines the loopback interface represents a genuine logical interface, commonly named `lo`. On Windows systems the "interface" may not represent an actual operating-system entity, but this is of practical concern only to those interested in "sniffing" all loopback traffic; packets sent to the loopback address are still delivered as expected.

Workstations often have special other interfaces as well. Most recent versions of Microsoft Windows have a Teredo Tunneling pseudo-interface and an Automatic Tunneling pseudo-interface; these are both intended (when activated) to support IPv6 connectivity when the local ISP supports only IPv4. The Teredo protocol is documented in **RFC 4380**.

When VPN connections are created, as in *3.1 Virtual Private Networks*, each end of the logical connection typically terminates at a virtual interface (one of these is labeled `tun0` in the diagram of *3.1 Virtual Private Networks*). These virtual interfaces appear, to the systems involved, to be attached to a point-to-point link that leads to the other end.

When a computer hosts a virtual machine, there is almost always a virtual network to connect the host and virtual systems. The host will have a virtual interface to connect to the virtual network. The host may act as a NAT router for the virtual machine, "hiding" that virtual machine behind its own IP address, or it may act as an Ethernet switch, in which case the virtual machine will need an additional public IP address.

What's My IP Address?

This simple-seeming question is in fact not very easy to answer, if by "my IP address" one means the IP address assigned to the interface that connects directly to the Internet. One strategy is to find the address of the default router, and then iterate through all interfaces (*eg* with the Java `NetworkInterface` class) to find an IP address with a matching network prefix; a Python3 example of this approach appears in *18.5.1 Multicast Programming*. Unfortunately, finding the default router (to identify the primary interface) is hard to do in an OS-independent way, and even then this approach can fail if the Wi-Fi and Ethernet interfaces both are assigned IP addresses on the same network, but only one is actually connected.

Routers always have at least two interfaces on two separate IP networks. Generally this means a separate IP address for each interface, though some point-to-point interfaces can be used without being assigned any IP address (*7.12 Unnumbered Interfaces*).

7.2.1 Multihomed hosts

A non-router host with multiple non-loopback network interfaces is often said to be **multihomed**. Many laptops, for example, have both an Ethernet interface and a Wi-Fi interface. Both of these can be used simultaneously, with different IP addresses assigned to each. On residential networks the two interfaces will often be on the same IP network (*eg* the same bridged Wi-Fi/Ethernet LAN); at more security-conscious sites the Ethernet and Wi-Fi interfaces are often on quite different IP networks (though see *7.9.5 ARP and multihomed hosts*).

Multiple physical interfaces are not actually needed here; it is usually possible to assign multiple IP addresses to a *single* interface. Sometimes this is done to allow two IP networks (two distinct prefixes) to share a single physical LAN; in this case the interface would be assigned one IP address for each IP network. Other times a single interface is assigned multiple IP addresses on the same IP network; this is often done so that one physical machine can act as a server (*eg* a web server) for multiple distinct IP addresses corresponding to multiple distinct domain names.

Multihoming raises some issues with packets addressed to one interface, A, with IP address A_{IP}, but which arrive via another interface, B, with IP address B_{IP}. Strictly speaking, such arriving packets should be discarded unless the host is promoted to functioning as a router. In practice, however, the strict interpretation often causes problems; a typical user understanding is that the IP address A_{IP} should work to reach the host even if the physical connection is to interface B. A related issue is whether the host receiving such a packet addressed to A_{IP} on interface B is allowed to send its *reply* with source address A_{IP}, even though the reply must be sent via interface B.

RFC 1122, §3.3.4, defines two alternatives here:

- The **Strong End-System** model: IP addresses – incoming and outbound – must match the physical interface.
- The **Weak End-System** model: A match is not required: interface B can accept packets addressed to A_{IP}, and send packets with source address A_{IP}.

Linux systems generally use the weak model by default. See also *7.9.5 ARP and multihomed hosts*.

While it is important to be at least vaguely aware of the special cases that multihoming presents, we emphasize again that in most ordinary contexts each end-user workstation has one IP address that corresponds to a LAN connection.

7.3 Special Addresses

A few IPv4 addresses represent special cases.

While the standard IPv4 loopback address is 127.0.0.1, any IPv4 address beginning with 127 can serve as a loopback address. Logically they all represent the current host. Most hosts are configured to resolve the name "localhost" to 127.0.0.1. However, any loopback address – *eg* 127.255.37.59 – should work, *eg* with `ping`. For an example using 127.0.**1**.0, see *7.8 DNS*.

Private addresses are IPv4 addresses intended only for site internal use, *eg* either behind a NAT firewall or intended to have no Internet connectivity at all. If a packet shows up at any non-private router (*eg* at an ISP router), with a private IPv4 address as either source or destination address, the packet should be dropped. Three standard private-address blocks have been defined:

- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

The last block is the one from which addresses are most commonly allocated by DHCP servers (*7.10.1 NAT, DHCP and the Small Office*) built into NAT routers.

Broadcast addresses are a special form of IPv4 address intended to be used in conjunction with LAN-layer broadcast. The most common forms are "broadcast to this network", consisting of all 1-bits, and "broadcast to network D", consisting of D's network-address bits followed by all 1-bits for the host bits. If you try to send a packet to the broadcast address of a remote network D, the odds are that some router involved will refuse to forward it, and the odds are even higher that, once the packet arrives at a router actually on network D, that router will refuse to broadcast it. Even addressing a broadcast to one's own network will fail if the underlying LAN does not support LAN-level broadcast (*eg* ATM).

The highly influential early Unix implementation Berkeley 4.2 BSD used 0-bits for the broadcast bits, instead of 1's. As a result, to this day host bits cannot be all 1-bits or all 0-bits in order to avoid confusion with the IPv4 broadcast address. One consequence of this is that a Class C network has 254 usable host addresses, not 256.

7.3.1 Multicast addresses

Finally, **IPv4 multicast addresses** remain as the last remnant of the Class A/B/C strategy: multicast addresses are Class D, with first byte beginning 1110 (meaning that the first byte is, in decimal, 224-239). Multicasting means delivering to a specified *set* of addresses, preferably by some mechanism more efficient than sending to each address individually. A reasonable goal of multicast would be that no more than one copy of the multicast packet traverses any given link.

Support for IPv4 multicast requires considerable participation by the backbone routers involved. For example, if hosts A, B and C each connect to different interfaces of router R1, and A wishes to send a multicast

packet to B and C, then it is up to R1 to receive the packet, figure out that B and C are the intended recipients, and forward the packet *twice*, once for B's interface and once for C's. R1 must also keep track of what hosts have joined the **multicast group** and what hosts have left. Due to this degree of router participation, backbone router support for multicasting has not been entirely forthcoming. A discussion of IPv4 multicasting appears in *20 Quality of Service*.

7.4 Fragmentation

If you are trying to interconnect two LANs (as IP does), what else might be needed besides Routing and Addressing? IPv4 (and IPv6) explicitly assumes all packets are composed on 8-bit bytes (something not universally true in the early days of IP; to this day the RFCs refer to "octets" to emphasize this requirement). IP also defines bit-order within a byte, and it is left to the networking hardware to translate properly. Neither byte size nor bit order, therefore, can interfere with packet forwarding.

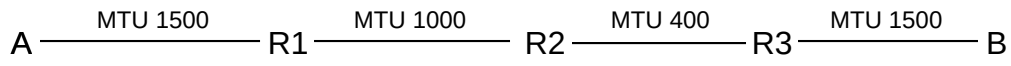
There is one more feature IPv4 must provide, however, if the goal is universal connectivity: it must accommodate networks for which the maximum packet size, or **Maximum Transfer Unit**, MTU, is smaller than the packet that needs forwarding. Otherwise, if we were using IPv4 to join Token Ring (MTU = 4kB, at least originally) to Ethernet (MTU = 1500B), the token-ring packets might be too large to deliver to the Ethernet side, or to traverse an Ethernet backbone *en route* to another Token Ring. (Token Ring, in its day, did commonly offer a configuration option to allow Ethernet interoperability.)

So, IPv4 must support fragmentation, and thus also reassembly. There are two potential strategies here: **per-link** fragmentation and reassembly, where the reassembly is done at the opposite end of the link (as in ATM), and **path** fragmentation and reassembly, where reassembly is done at the far end of the path. The latter approach is what is taken by IPv4, partly because intermediate routers are too busy to do reassembly (this is as true today as it was in 1981 when **RFC 791** was published), partly because there is no absolute guarantee that all fragments will go to the same next-hop router, and partly because IPv4 fragmentation has always been seen as the strategy of last resort.

An IPv4 sender is supposed to use a different value for the **IDENT** field for different packets, at least up until the field wraps around. When an IPv4 datagram is fragmented, the fragments keep the same IDENT field, so this field in effect indicates which fragments belong to the same packet.

After fragmentation, the **Fragment Offset** field marks the start position of the data portion of this fragment within the data portion of the original IPv4 packet. Note that the start position can be a number up to 2^{16}, the maximum IPv4 packet length, but the FragOffset field has only 13 bits. This is handled by requiring the data portions of fragments to have sizes a multiple of 8 (three bits), and left-shifting the FragOffset value by 3 bits before using it.

As an example, consider the following network, where MTUs are excluding the LAN header:



Suppose A addresses a packet of 1500 bytes to B, and sends it via the LAN to the first router R1. The packet contains 20 bytes of IPv4 header and 1480 of data.

R1 fragments the original packet into two packets of sizes 20+976 = 996 and 20+504=544. Having 980

bytes of payload in the first fragment would fit, but violates the rule that the sizes of the data portions be divisible by 8. The first fragment packet has FragOffset = 0; the second has FragOffset = 976.

R2 *re*fragments the first fragment into three packets as follows:

- first: size = 20+376=396, FragOffset = 0
- second: size = 20+376=396, FragOffset = 376
- third: size = 20+224 = 244 (note 376+376+224=976), FragOffset = 752.

R2 refragments the second fragment into two:

- first: size = 20+376 = 396, FragOffset = 976+0 = 976
- second: size = 20+128 = 148, FragOffset = 976+376=1352

R3 then sends the fragments on to B, without reassembly.

Note that it would have been slightly more efficient to have fragmented into four fragments of sizes 376, 376, 376, and 352 in the beginning. Note also that the packet format is designed to handle fragments of different sizes easily. The algorithm is based on multiple fragmentation with reassembly only at the final destination.

Each fragment has its IPv4-header Total Length field set to the length of that fragment.

We have not yet discussed the three flag bits. The first bit is reserved, and must be 0. The second bit is the **Don't Fragment**, or DF, bit. If it is set to 1 by the sender then a router must *not* fragment the packet and must drop it instead; see *12.13 Path MTU Discovery* for an application of this. The third bit is set to 1 for all fragments *except* the final one (this bit is thus set to 0 if no fragmentation has occurred). The third bit tells the receiver where the fragments stop.

The receiver must take the arriving fragments and **reassemble** them into a whole packet. The fragments may not arrive in order – unlike in ATM networks – and may have unrelated packets interspersed. The reassembler must identify when different arriving packets are fragments of the same original, and must figure out how to reassemble the fragments in the correct order; both these problems were essentially trivial for ATM.

Fragments are considered to belong to the same packet if they have the same IDENT field and also the same source and destination addresses and same protocol.

As all fragment sizes are a multiple of 8 bytes, the receiver can keep track of whether all fragments have been received with a bitmap in which each bit represents one 8-byte fragment chunk. A 1 kB packet could have up to 128 such chunks; the bitmap would thus be 16 bytes.

If a fragment arrives that is part of a new (and fragmented) packet, a buffer is allocated. While the receiver cannot know the final size of the buffer, it can usually make a reasonable guess. Because of the FragOffset field, the fragment can then be stored in the buffer in the appropriate position. A new bitmap is also allocated, and a **reassembly timer** is started.

As subsequent fragments arrive, not necessarily in order, they too can be placed in the proper buffer in the proper position, and the appropriate bits in the bitmap are set to 1.

If the bitmap shows that all fragments have arrived, the packet is sent on up as a completed IPv4 packet. If, on the other hand, the reassembly timer expires, then all the pieces received so far are discarded.

TCP connections usually engage in **Path MTU Discovery**, and figure out the largest packet size they can send that will *not* entail fragmentation (*12.13 Path MTU Discovery*). But it is not unusual, for example, for UDP protocols to use fragmentation, especially over the short haul. In the Network File System (NFS) protocol, for example, UDP is used to carry 8 kB disk blocks. These are often sent as a single 8+ kB IPv4 packet, fragmented over Ethernet to five full packets and a fraction. Fragmentation works reasonably well here because most of the time the packets do not leave the Ethernet they started on. Note that this is an example of fragmentation done by the *sender*, not by an intermediate router.

Finally, any given IP link may provide its own link-layer fragmentation and reassembly; we saw in *3.5.1 ATM Segmentation and Reassembly* that ATM does just this. Such link-layer mechanisms are, however, generally invisible to the IP layer.

7.5 The Classless IP Delivery Algorithm

Recall from Chapter 1 that any IPv4 address can be divided into a net portion IP_{net} and a host portion IP_{host}; the division point was determined by whether the IPv4 address was a Class A, a Class B, or a Class C. We also indicated in Chapter 1 that the division point was not always so clear-cut; we now present the delivery algorithm, for both hosts and routers, that does *not* assume a globally predeclared division point of the input IPv4 address into net and host portions. We will, for the time being, punt on the question of forwarding-table lookup and assume there is a `lookup()` method available that, when given a destination address, returns the next_hop neighbor.

Instead of class-based divisions, we will assume that each of the IPv4 addresses assigned to a node's interfaces is configured with an associated length of the network prefix; following the slash notation of *1.10 IP - Internet Protocol*, if B is an address and the prefix length is $k = k_B$ then the prefix itself is B/k. As usual, an ordinary host may have only one IP interface, while a router will always have multiple interfaces.

Let D be the given IPv4 destination address; we want to decide if D is **local** or **nonlocal**. The host or router involved may have multiple IP interfaces, but for each interface the length of the network portion of the address will be known. For each network address B/k assigned to one of the host's interfaces, we compare the first k bits of B and D; that is, we ask if D **matches** B/k.

- If one of these comparisons yields a match, delivery is **local**; the host delivers the packet to its final destination via the LAN connected to the corresponding interface. This means looking up the LAN address of the destination, if applicable, and sending the packet to that destination via the interface.
- If there is no match, delivery is **nonlocal**, and the host passes D to the `lookup()` routine of the forwarding table and sends to the associated next_hop (which must represent a physically connected neighbor). It is now up to `lookup()` routine to make any necessary determinations as to how D might be split into D_{net} and D_{host}; the split *cannot* be made outside of `lookup()`.

The forwarding table is, abstractly, a set of network addresses – now also with lengths – each of the form B/k, with an associated next_hop destination for each. The `lookup()` routine will, in principle, compare D with each table entry B/k, looking for a match (that is, equality of the first $k = k_B$ bits). As with the local-delivery interfaces check above, the net/host division point (that is, k) will come from the table entry; it will not be inferred from D or from any other information borne by the packet. There is, in fact, no place in the IPv4 header to store a net/host division point, and furthermore different routers along the path may use different values of k with the same destination address D. Routers receive the prefix length /k for a destination B/k as part of the process by which they receive ⟨destination,next_hop⟩ pairs; see *9 Routing-Update Algorithms*.

In *10 Large-Scale IP Routing* we will see that in some cases multiple matches in the forwarding table may exist, *eg* 147.0.0.0/8 and 147.126.0.0/16. The **longest-match** rule will be introduced for such cases to pick the best match.

Here is a simple example for a router with immediate neighbors A-E:

destination	next_hop
10.3.0.0/16	A
10.4.1.0/24	B
10.4.2.0/24	C
10.4.3.0/24	D
10.3.37.0/24	E

The IPv4 addresses 10.3.67.101 and 10.3.59.131 both route to A. The addresses 10.4.1.101, 10.4.2.157 and 10.4.3.233 route to B, C and D respectively. Finally, 10.3.37.103 matches both A and E, but the E match is longer so the packet is routed that way.

The forwarding table may also contain a **default** entry for the next_hop, which it may return in cases when the destination D does not match any known network. We take the view here that returning such a default entry is a valid result of the routing-table `lookup()` operation, rather than a third option to the algorithm above; one approach is for the default entry to be the next_hop corresponding to the destination 0.0.0.0/0, which does indeed match everything (use of this would definitely require the above longest-match rule, though).

Default routes are hugely important in keeping leaf forwarding tables small. Even backbone routers sometimes expend considerable effort to keep the network address prefixes in their forwarding tables as short as possible, through consolidation.

At a site with a single ISP and with no Internet customers (that is, which is not itself an ISP for others), the top-level forwarding table usually has a single external route: its default route to its ISP. If a site has more than one ISP, however, the top-level forwarding table can expand in a hurry. For example, Internet2 is a consortium of research sites with very-high-bandwidth internal interconnections, acting as a sort of "parallel Internet". Before Internet2, Loyola's top-level forwarding table had the usual single external default route. After Internet2, we in effect had a second ISP and had to divide traffic between the commercial ISP and the Internet2 ISP. The default route still pointed to the commercial ISP, but the top-level forwarding table now had to have an entry for every individual Internet2 site, so that traffic to any of these sites would be forwarded via the Internet2 ISP. See exercise 5.0.

Routers may also be configured to allow passing quality-of-service information to the `lookup()` method, as mentioned in Chapter 1, to support different routing paths for different kinds of traffic (*eg* bulk file-transfer versus real-time).

For a modest exception to the local-delivery rule described here, see below in *7.12 Unnumbered Interfaces*.

7.6 IPv4 Subnets

Subnets were the first step away from Class A/B/C routing: a large network (*eg* a class A or B) could be divided into smaller IPv4 networks called subnets. Consider, for example, a typical Class B network such as Loyola University's (originally 147.126.0.0/16); the underlying assumption is that any packet can be

delivered via the underlying LAN to any internal host. This would require a rather large LAN, and would require that a single physical LAN be used throughout the site. What if our site has more than one physical LAN? Or is really too big for one physical LAN? It did not take long for the IP world to run into this problem.

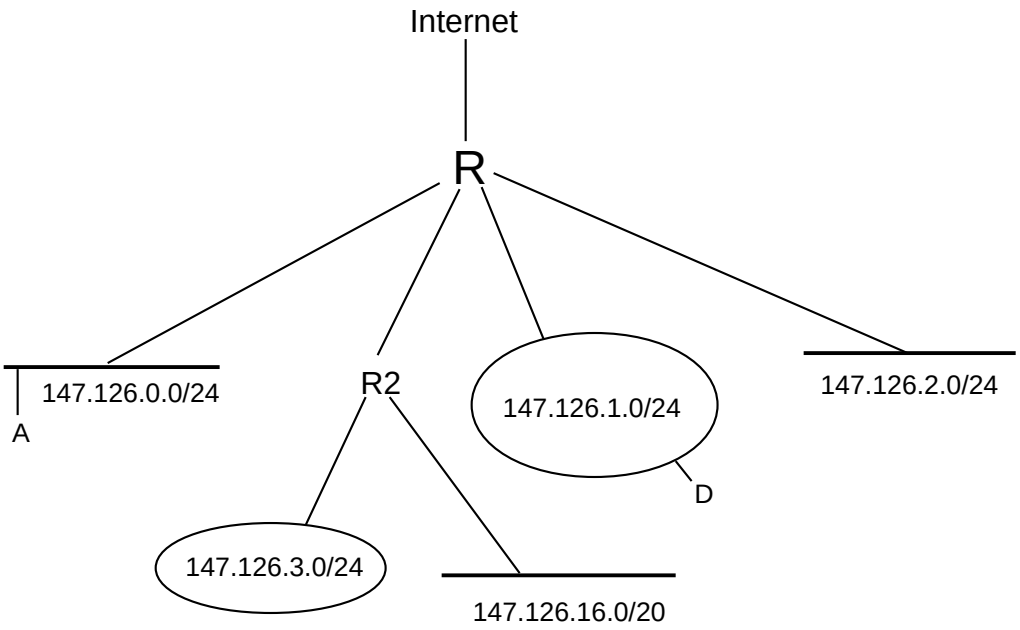
Subnets were first proposed in **RFC 917**, and became official with **RFC 950**.

Getting a separate IPv4 network prefix for each subnet is bad for routers: the backbone forwarding tables now must have an entry for every subnet instead of just for every site. What is needed is a way for a site to appear to the outside world as a single IP network, but for further IP-layer routing to be supported *inside* the site. This is what subnets accomplish.

Subnets introduce **hierarchical routing**: first we route to the primary network, then inside that site we route to the subnet, and finally the last hop delivers to the host.

Routing with subnets involves in effect moving the IP_{net} division line rightward. (Later, when we consider CIDR, we will see the complementary case of moving the division line to the left.) For now, observe that moving the line rightward within a site does not affect the outside world at all; outside routers are not even aware of site-internal subnetting.

In the following diagram, the outside world directs traffic addressed to 147.126.0.0/16 to the router R. Internally, however, the site is divided into subnets. The idea is that traffic from 147.126.1.0/24 to 147.126.2.0/24 is routed, not switched; the two LANs involved may not even be compatible. Most of the subnets shown are of size /24, meaning that the third byte of the IPv4 address has become part of the network portion of the subnet's address; one /20 subnet is also shown. **RFC 950** would have disallowed the subnet with third byte 0, but having 0 for the subnet bits generally does work.



What we want is for the internal routing to be based on the extended network prefixes shown, while externally continuing to use only the single routing entry for 147.126.0.0/16.

To implement subnets, we divide the site's IPv4 network into some combination of physical LANs – the subnets –, and assign each a **subnet address**: an IPv4 network address which has the *site's* IPv4 network

address as prefix. To put this more concretely, suppose the site's IPv4 network address is A, and consists of n network bits (so the site address may be written with the slash notation as A/n); in the diagram above, A/n = 147.126.0.0/16. A subnet address is an IPv4 network address B/k such that:

- The address B/k is within the site: the first n bits of B are the same as A/n's
- B/k extends A/n: k⩾n

An example B/k in the diagram above is 147.126.1.0/24. (There is a slight simplification here in that subnet addresses do not absolutely *have* to be prefixes; see below.)

We now have to figure out how packets will be routed to the correct subnet. For incoming packets we could set up some proprietary protocol at the entry router to handle this. However, the more complicated situation is all those existing internal hosts that, under the class A/B/C strategy, would still believe they can deliver via the LAN to any site host, when in fact they can now only do that for hosts on their own subnet. We need a more general solution.

We proceed as follows. For each subnet address B/k, we create a **subnet mask** for B consisting of k 1-bits followed by enough 0-bits to make a total of 32. We then make sure that every host and router in the site knows the subnet mask for every one of its *interfaces*. Hosts usually find their subnet mask the same way they find their IP address (by static configuration if necessary, but more likely via DHCP, below).

Hosts and routers now apply the IP delivery algorithm of the previous section, with the proviso that, if a subnet mask for an interface is present, then the subnet mask is used to determine the number of address bits rather than the Class A/B/C mechanism. That is, we determine whether a packet addressed to destination D is deliverable locally via an interface with subnet address B/k and corresponding mask M by comparing D&M with B&M, where **&** represents bitwise AND; if the two match, the packet is local. This will generally involve a match of *more* bits than if we used the Class A/B/C strategy to determine the network portion of addresses D and B.

As stated previously, given an address D with no other context, we will *not* be able to determine the network/host division point in general (*eg* for outbound packets). However, that division point is not in fact what we need. All that *is* needed is a way to tell if a given destination host address D belongs to the current subnet, say B; that is, we need to compare the first k bits of D and B where k is the (known) length of B.

In the diagram above, the subnet mask for the /24 subnets would be 255.255.255.0; bitwise ANDing any IPv4 address with the mask is the same as extracting the first 24 bits of the IPv4 address, that is, the subnet portion. The mask for the /20 subnet would be 255.255.240.0 (240 in binary is 1111 0000).

In the diagram above none of the subnets overlaps or conflicts: the subnets 147.126.0.0/24 and 147.126.1.0/24 are disjoint. It takes a little more effort to realize that 147.126.16.0/20 does not overlap with the others, but note that an IPv4 address matches this network prefix only if the first four bits of the third byte are 0001, so the third byte itself ranges from decimal 32 to decimal 63 = binary 0001 1111.

Note also that if host A = 147.126.0.1 wishes to send to destination D = 147.126.1.1, and A is *not* subnet-aware, then delivery will fail: A will infer that the interface is a Class B, and therefore compare the first two bytes of A and D, and, finding a match, will attempt direct LAN delivery. But direct delivery is now likely impossible, as the subnets are not joined by a switch. Only with the subnet mask will A realize that its network is 147.126.0.0/**24** while D's is 147.126.1.0/24 and that these are not the same. A *would* still be able to send packets to its own subnet. In fact A would still be able to send packets to the outside world: it would realize that the destination in that case does not match 147.126.0.0/16 and will thus forward to its router. Hosts on other subnets would be the only unreachable ones.

Properly, the subnet address is the entire prefix, *eg* 147.126.65.0/24. However, it is often convenient to identify the subnet address with just those bits that represent the extension of the site IPv4-network address; we might thus say casually that the subnet address here is 65.

The class-based IP-address strategy allowed any host anywhere on the Internet to properly separate any address into its net and host portions. With subnets, this division point is now allowed to vary; for example, the address 147.126.65.48 divides into 147.126 | 65.48 outside of Loyola, but into 147.126.65 | 48 inside. This means that the net-host division is no longer an absolute property of addresses, but rather something that depends on where the packet is on its journey.

Technically, we also need the requirement that given any two subnet addresses of different, disjoint subnets, neither is a proper prefix of the other. This guarantees that if A is an IP address and B is a subnet address with mask M (so B = B&M), then A&M = B implies A does not match any other subnet. Regardless of the net/host division rules, we cannot possibly allow subnet 147.126.16.0/20 to represent one LAN while 147.126.16.0/24 represents another; the second subnet address block is a subset of the first. (We *can*, and sometimes do, allow the first LAN to correspond to everything in 147.126.16.0/20 that is not also in 147.126.16.0/24; this is the longest-match rule.)

The strategy above is actually a slight simplification of what the subnet mechanism actually allows: subnet address bits do not in fact have to be contiguous, and masks do not have to be a series of 1-bits followed by 0-bits. The mask can be *any* bit-mask; the subnet address bits are by definition those where there is a 1 in the mask bits. For example, we could at a Class-B site use the *fourth* byte as the subnet address, and the *third* byte as the host address. The subnet mask would then be 255.255.0.255. While this generality was once sometimes useful in dealing with "legacy" IPv4 addresses that could not easily be changed, life is simpler when the subnet bits precede the host bits.

7.6.1 Subnet Example

As an example of having different subnet masks on different interfaces, let us consider the division of a class-C network into subnets of size 70, 40, 25, and 20. The subnet addresses will of necessity have different lengths, as there is not room for four subnets each able to hold 70 hosts.

- A: size 70
- B: size 40
- C: size 25
- D: size 20

Because of the different subnet-address lengths, division of a local IPv4 address LA into net versus host on subnets cannot be done in isolation, without looking at the host bits. However, that division is not in fact what we need. All that is needed is a way to tell if the local address LA belongs to a given subnet, say B; that is, we need to compare the first n bits of LA and B, where n is the length of B's subnet mask. We do this by comparing LA&M to B&M, where M is the mask corresponding to n. LA&M is not necessarily the same as LA_{net}, if LA actually belongs to one of the other subnets. However, if LA&M = B&M, then LA must belong subnet B, in which case LA&M is in fact LA_{net}.

We will assume that the site's IPv4 network address is 200.0.0.0/24. The first three bytes of each subnet address must match 200.0.0. Only some of the bits of the fourth byte will be part of the subnet address, so

we will switch to binary for the last byte, and use both the /n notation (for total number of subnet bits) and also add a vertical bar | to mark the separation between subnet and host.

Example: 200.0.0.10 | 00 0000 / 26

Note that this means that the 0-bit following the 1-bit in the fourth byte is "significant" in that for a subnet to match, it must match this 0-bit exactly. The remaining six 0-bits are part of the host portion.

To allocate our four subnet addresses above, we start by figuring out just how many host bits we need in each subnet. Subnet sizes are always powers of 2, so we round up the subnets to the appropriate size. For subnet A, this means we need 7 host bits to accommodate 2^7 = 128 hosts, and so we have a single bit in the fourth byte to devote to the subnet address. Similarly, for B we will need 6 host bits and will have 2 subnet bits, and for C and D we will need 5 host bits each and will have 8-5=3 subnet bits.

We now start choosing non-overlapping subnet addresses. We have one bit in the fourth byte to choose for A's subnet; rather arbitrarily, let us choose this bit to be 1. This means that *every other subnet address* must have a 0 in the first bit position of the fourth byte, or we would have ambiguity.

Now for B's subnet address. We have two bits to work with, and the first bit must be 0. Let us choose the second bit to be 0 as well. If the fourth byte begins 00, the packet is part of subnet B, and the subnet addresses for C and D must therefore *not* begin 00.

Finally, we choose subnet addresses for C and D to be 010 and 011, respectively. We thus have

subnet	size	address bits in fourth byte	host bits in 4th byte	decimal range
A	128	1	7	128-255
B	64	00	6	0-63
C	32	010	5	64-95
D	32	011	5	96-127

As desired, none of the subnet addresses in the third column is a prefix of any other subnet address.

The end result of all of this is that routing is now **hierarchical**: we route on the site IP address to get to a site, and then route on the subnet address within the site.

7.6.2 Links between subnets

Suppose the Loyola CS department subnet (147.126.65.0/24) and a department at some other site, we will say 147.100.100.0/24, install a private link. How does this affect routing?

Each department router would add an entry for the other subnet, routing along the private link. Traffic addressed to the other subnet would take the private link. All other traffic would go to the default router. Traffic from the remote department to 147.126.6**4**.0/24 would take the long route, and Loyola traffic to 147.100.10**1**.0/24 would take the long route.

Subnet anecdote

A long time ago I was responsible for two hosts, abel and borel. One day I was informed that machines in computer lab 1 at the other end of campus could not reach borel, though they could reach abel. Machines

in lab 2, *adjacent* to lab 1, however, could reach both borel and abel just fine. What was the difference?

It turned out that borel had a bad (/16 instead of /24) subnet mask, and so it was attempting local delivery to the labs. This *should* have meant it could reach neither of the labs, as both labs were on a different subnet from my machines; I was still perplexed. After considerably more investigation, it turned out that between abel/borel and the lab building was a **bridge-router**: a hybrid device that properly routed subnet traffic at the IP layer, but which also forwarded Ethernet packets directly, the latter feature apparently for the purpose of backwards compatibility. Lab 2 was connected directly to the bridge-router and thus appeared to be on the same LAN as borel, despite the apparently different subnet; lab 1 was connected to its own router R1 which in turn connected to the bridge-router. Lab 1 was thus, at the LAN level, isolated from abel and borel.

Moral 1: Switching and routing are both great ideas, alone. But switching at one layer mixed with routing at another is not.

Moral 2: Test thoroughly! The reason the problem wasn't noticed earlier was that previously borel communicated only with other hosts on its own subnet and with hosts outside the university entirely. Both of these worked with the bad subnet mask; it was different-subnet local hosts that were the problem.

How would nearby subnets at either endpoint decide whether to use the private link? Classical link-state or distance-vector theory (*9 Routing-Update Algorithms*) requires that they be able to compare the private-link route with the going-around-the-long-way route. But this requires a global picture of relative routing costs, which, as we shall see, almost certainly does not exist. The two departments are in different routing domains; if neighboring subnets at either end want to use the private link, then manual configuration is likely the only option.

7.6.3 Subnets versus Switching

A frequent network design question is whether to have many small subnets or to instead have just a few (or even only one) larger subnet. With multiple small subnets, IP **routing** would be used to interconnect them; the use of larger subnets would replace much of that routing with LAN-layer communication, likely Ethernet **switching**. Debates on this route-versus-switch question have gone back and forth in the networking community, with one aphorism summarizing a common view:

> Switch when you can, route when you must

This aphorism reflects the idea that switching is faster, cheaper and easier to configure, and that subnet boundaries should be drawn only where "necessary".

Ethernet switching equipment is indeed generally cheaper than routing equipment, for the same overall level of features and reliability. And traditional switching requires relatively little configuration, while to implement subnets not only must the subnets be created by hand but one must also set up and configure the routing-update protocols. However, the price difference between switching and routing is not always significant in the big picture, and the configuration involved is often straightforward.

Somewhere along the way, however, switching has acquired a reputation – often deserved – for being *faster* than routing. It is true that routers have more to do than switches: they must decrement TTL, update the header checksum, and attach a new LAN header. But these things are relatively minor: a larger reason many routers are slower than switches may simply be that they are inevitably *asked to serve as firewalls*. This means "deep inspection" of every packet, *eg* comparing every packet to each of a large number of firewall

rules. The firewall may also be asked to keep track of connection state. All this drives down the forwarding rate, as measured in packets-per-second.

Traditional switching scales remarkably well, but it does have limitations. First, broadcast packets must be forwarded throughout a switched network; they do not, however, pass to different subnets. Second, LAN networks do not like redundant links (that is, loops); while one can rely on the spanning-tree algorithm to eliminate these, that algorithm too becomes less efficient at larger scales.

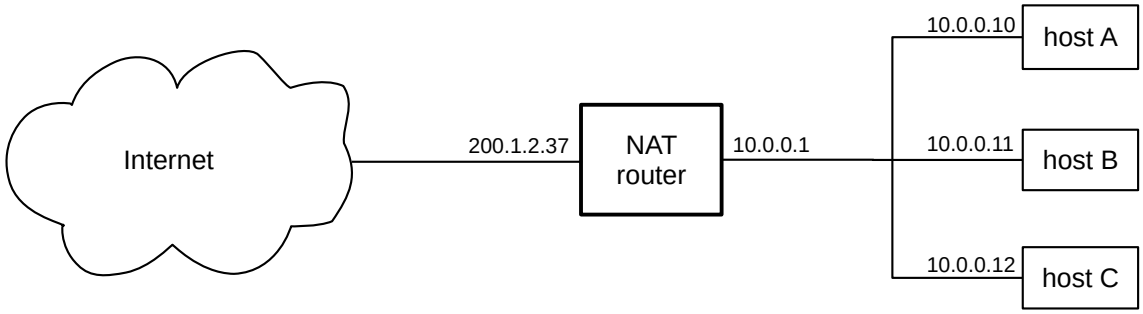
The rise of software-defined networking (*2.8 Software-Defined Networking*) has blurred the distinction between routing and switching. The term "Layer 3 switch" is sometimes used to describe routers that in effect do not support all the usual firewall bells and whistles. These are often SDN Ethernet switches (*2.8 Software-Defined Networking*) that are making forwarding decisions based on the contents of the IP header. Such streamlined switch/routers may also be able to do most of the hard work in specialized hardware, another source of speedup.

But SDN can do much more than IP-layer forwarding, by taking advantage of site-specific layout information. One application, of a switch hierarchy for traffic entering a datacenter, appears in *2.8.1 OpenFlow Switches*. Other SDN applications include enabling Ethernet topologies with loops, offloading large-volume flows to alternative paths, and implementing policy-based routing as in *9.6 Routing on Other Attributes*. Some SDN solutions involve site-specific programming, but others work more-or-less out of the box. Locations with switch-versus-route issues are likely to turn increasingly to SDN in the future.

7.7 Network Address Translation

What do you do if your ISP assigns to you a single IPv4 address and you have two computers? The solution is Network Address Translation, or **NAT**. NAT's ability to "multiplex" an arbitrarily large number of individual hosts behind a single IPv4 address (or small number of addresses) makes it an important tool in the conservation of IPv4 addresses. It also, however, enables an important form of firewall-based security. It is documented in **RFC 3022**, where this is called NAPT, or Network Address **Port** Translation.

The basic idea is that, instead of assigning each host at a site a publicly visible IPv4 address, just one such address is assigned to a special device known as a NAT router. A NAT router sold for residential or small-office use is commonly simply called a "router", or (somewhat more precisely) a "residential gateway". One side of the NAT router connects to the Internet; the other connects to the site's internal network. Hosts on the internal network are assigned private IP addresses (*7.3 Special Addresses*), typically of the form or 192.168.x.y or 10.x.y.z. Connections to internal hosts that originate in the outside world are banned. When an internal machine wants to connect to the outside, the NAT router intercepts the connection, and forwards the connection's packets after rewriting the source address to make it appear they came from the NAT router's own IP address, shown below as 200.1.2.37.



The remote machine responds, sending its responses to the NAT router's public IPv4 address. The NAT router remembers the connection, having stored the connection information in a special forwarding table, and forwards the data to the correct internal host, rewriting the destination-address field of the incoming packets.

The NAT forwarding table also includes port numbers. That way, if two internal hosts attempt to connect to the same external host, the NAT router can tell which packets belong to which. For example, suppose internal hosts A and B each connect from port 3000 to port 80 on external hosts S and T, respectively. Here is what the NAT forwarding table might look like. No columns for the NAT router's own IPv4 addresses are needed; we shall let NR denote the router's external address.

remote host	remote port	outside source port	inside host	inside port
S	80	3000	A	3000
T	80	3000	B	3000

A packet to S from ⟨A,3000⟩ would be rewritten so that the source was ⟨NR,3000⟩. A packet from ⟨S,80⟩ addressed to ⟨NR,3000⟩ would be rewritten and forwarded to ⟨A,3000⟩. Similarly, a packet from ⟨T,80⟩ addressed to ⟨NR,3000⟩ would be rewritten and forwarded to ⟨B,3000⟩; the NAT table takes into account the source host and port as well as the destination.

Sometimes it is necessary for the NAT router to rewrite the internal-side port number as well; this happens if two internal hosts want to connect, each from the *same* port, to the same external host and port. For example, suppose B now opens a connection to ⟨S,80⟩, also from inside port 3000. This time the NAT router must remap the port number, because that is the only way to distinguish between packets from ⟨S,80⟩ back to A and to B. With B's second connection's internal port remapped from 3000 to 3001, the new table is

remote host	remote port	outside source port	inside host	inside port
S	80	3000	A	3000
T	80	3000	B	3000
S	80	3001	B	3000

The NAT router does not create TCP connections between itself and the external hosts; it simply forwards packets (with rewriting). The connection endpoints are still the external hosts S and T and the internal hosts A and B. However, NR might very well *monitor* the TCP connections to know when they have closed, and so can be removed from the table. For UDP connections, NAT routers typically remove the forwarding entry after some period of inactivity; see *11 UDP Transport*, exercise 14.0.

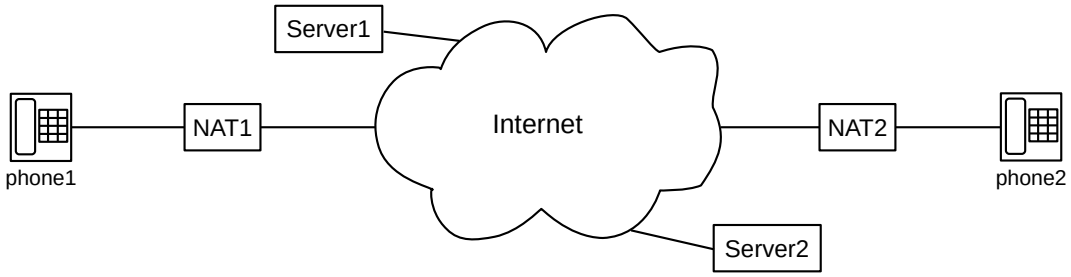
NAT still works for *some* traffic without port numbers, such as network pings, though the above table is then not quite the whole story. See *7.11 Internet Control Message Protocol*.

Done properly, NAT improves the security of a site, by making it impossible for an external host to probe or connect to any of the internal hosts. While this firewall feature is of great importance, essentially the same effect can be achieved without address translation, and *with* public IPv4 addresses for all internal hosts, by having the router refuse to forward incoming packets that are not part of existing connections. The router still needs to maintain a table like the NAT table above, in order to recognize such packets. The address translation itself, in other words, is not the source of the firewall security. That said, it is hard for a NAT router to "fail open"; *ie* to fail in a way that lets outside connections in. It is much easier for a non-NAT firewall to fail open.

For the common residential form of NAT router, see *7.10.1 NAT, DHCP and the Small Office*.

7.7.1 NAT Problems

NAT router's refusal to allow inbound connections is a source of occasional frustration. We illustrate some of these frustrations here, using Voice-over-IP (VoIP) and the call-setup protocol SIP (**RFC 3261**). The basic strategy is that each phone is associated with a remote **phone server**. These phone servers, because they have to be able to accept incoming connections from anywhere, must not be behind NAT routers. The phones themselves, however, usually will be:



For phone1 to call phone2, phone1 first contacts Server1, which then contacts Server2. So far, all is well. The final step is for Server2 to contact phone2, which, however, cannot be done normally as NAT2 allows no inbound connections.

One common solution is for phone2 to maintain a persistent connection to Server2 (and ditto for phone1 and Server1). By having these persistent phone-to-server connections, we can arrange for the phone to ring on incoming calls.

As a second issue, somewhat particular to the SIP protocol, is that it is common for server and phone to prefer to use UDP port 5060 at *both* ends. For a single internal phone, it is likely that port 5060 will pass through without remapping, so the phone will appear to be connecting from the desired port 5060. However, if there are two phones inside (not shown above), one of them will appear to be connecting to the server *from* an alternative port. The solution here is to have the server tolerate such port remapping.

VoIP systems run into a much more serious problem with NAT, however. Once the call between phone1 and phone2 is set up, the servers would prefer to step out of the loop, and have the phones exchange voice packets directly. The SIP protocol was designed to handle this by having each phone report to its respective server the UDP socket (⟨IP address,port⟩ pair) it intends to use for the voice exchange; the servers then

report these phone sockets to each other, and from there to the opposite phones. This socket information is rendered incorrect by NAT, however, certainly the IP address and quite likely the port as well. If only one of the phones is behind a NAT firewall, it can initiate the voice connection to the other phone, but the other phone will see the voice packets arriving from a different socket than promised and will likely not recognize them as part of the call. If both phones are behind NAT firewalls, they will not be able to connect directly to one another at all. The common solution is for the VoIP server of a phone behind a NAT firewall to remain in the communications path, forwarding packets to its hidden partner. This works, but represents an unwanted server workload.

If a site wants to make it possible to allow external connections to hosts behind a NAT router or other firewall, one option is **tunneling**. This is the creation of a "virtual LAN link" that runs on top of a TCP connection between the end user and one of the site's servers; the end user can thus appear to be on one of the organization's internal LANs; see *3.1 Virtual Private Networks*. Another option is to "open up" a specific port: in essence, a static NAT-table entry is made connecting a specific port on the NAT router to a specific internal host and port (usually the same port). For example, all UDP packets to port 5060 on the NAT router might be forwarded to port 5060 on internal host A, even in the absence of any prior packet exchange. Gamers creating peer-to-peer game connections must also usually engage in some port-opening configuration. The Port Control Protocol (**RFC 6887**) is sometimes used for this.

NAT routers work very well when the communications model is of client-side TCP connections, originating from the inside and with public outside servers as destination. The NAT model works less well for peer-to-peer networking, as with the gamers above, where two computers, each behind a different NAT router, wish to establish a connection. Most NAT routers provide at least limited support for "opening" access to a given internal ⟨host,port⟩ socket, by creating a semi-permanent forwarding-table entry. See also *12.24 Exercises*, exercise 2.5.

NAT routers also often have trouble with UDP protocols, due to the tendency for such protocols to have the public server reply from a *different* port than the one originally contacted. For example, if host A behind a NAT router attempts to use TFTP (*11.2 Trivial File Transport Protocol, TFTP*), and sends a packet to port 69 of public server C, then C is likely to reply from some *new* port, say 3000, and this reply is likely to be dropped by the NAT router as there will be no entry there yet for traffic from ⟨C,3000⟩.

7.7.2 Middleboxes

Firewalls and NAT routers are sometimes classed as **middleboxes**: network devices that block, throttle or modify traffic beyond what is necessary for basic forwarding. Middleboxes play a very important role in network security, but they sometimes (as here with VoIP) break things. The word "middlebox" (versus "router" or "firewall") usually has a perjorative connotation; middleboxes have, in some circles, acquired a rather negative reputation.

NAT routers' interference with VoIP, above, is a direct consequence of their function: NAT handles connections from inside to outside quite well, but the NAT mechanism offers no support for connections from one inside to another inside. Sometimes, however, middleboxes block traffic when there is no technical reason to do so, simply because correct behavior has not been widely implemented. As an example, the SCTP protocol, *12.22.2 SCTP*, has seen very limited use despite some putative advantages over TCP, largely due to lack of NAT-router support. SCTP cannot be used by residential users because the middleboxes have not kept up.

A third category of middlebox-related problems is overzealous blocking in the name of security. SCTP

runs into this problem as well, though not quite as universally: a few routers simply drop all SCTP packets because they represent an "unknown" – and therefore suspect – type of traffic. There is a place for this block-by-default approach. If a datacenter firewall blocks all inbound TCP traffic except to port 80 (the HTTP port), and if SCTP is not being used within the datacenter intentionally, it is hard to argue against blocking all inbound SCTP traffic. But if the frontline router for home or office users blocks all *outbound* SCTP traffic, then the users cannot use SCTP.

A consequence of overzealous blocking is that it becomes much harder to introduce new protocols. If a new protocol is blocked for even a small fraction of potential users, it is just not worth the effort. See also the discussion at *12.22.4 QUIC Revisited*; the design of QUIC includes several elements to mitigate middlebox problems.

For another example of overzealous blocking by middleboxes, with the added element of spoofed TCP RST packets, see the sidebar at *14.8.3 Explicit Congestion Notification (ECN)*.

7.8 DNS

The **Domain Name System**, DNS, is an essential companion protocol to IPv4 (and IPv6); an overview can be found in **RFC 1034**. It is DNS that permits users the luxury of not needing to remember numeric IP addresses. Instead of 162.216.18.28, a user can simply enter intronetworks.cs.luc.edu, and DNS will take care of looking up the name and retrieving the corresponding address. DNS also makes it easy to move services from one server to another with a different IP address; as users will locate the service by **DNS name** and not by IP address, they do not need to be notified.

While DNS supports a wide variety of queries, for the moment we will focus on queries for IPv4 addresses, or so-called `A` records. The `AAAA` record type is used for IPv6 addresses, and, internally, the `NS` record type is used to identify the "name servers" that answer DNS queries.

While a workstation can use TCP/IP without DNS, users would have an almost impossible time finding anything, and so the core startup configuration of an Internet-connected workstation almost always includes the IP address of its DNS server (see *7.10 Dynamic Host Configuration Protocol (DHCP)* below for how startup configurations are often assigned).

Most DNS traffic today is over UDP, although a TCP option exists. Due to the much larger response sizes, TCP is often necessary for DNSSEC (*22.12 DNSSEC*).

DNS is **distributed**, meaning that each domain is responsible for maintaining its own **DNS servers** to translate names to addresses. (DNS, in fact, is a classic example of a highly distributed database where each node maintains a relatively small amount of data.) It is **hierarchical** as well; for the DNS name `intronetworks.cs.luc.edu` the levels of the hierarchy are

- **edu**: the **top-level domain** (TLD) for educational institutions in the US
- **luc**: Loyola University Chicago
- **cs**: The Loyola Computer Science Department
- **intronetworks**: a hostname associated to a specific IP address

The hierarchy of DNS names (that is, the set of all names and suffixes of names) forms a tree, but it is not only leaf nodes that represent individual hosts. In the example above, domain names luc.edu and cs.luc.edu happen to be valid hostnames as well.

The DNS hierarchy is in a great many cases not very deep, particularly for DNS names assigned to commercial websites. Such domain names are often simply the company name (or a variant of it) followed by the top-level domain (often `.com`). Still, internally most organizations have many individually named behind-the-scenes servers with three-level (or more) domain names; sometimes some of these can be identified by viewing the source of the web page and searching it for domain names.

Top-level domains are assigned by ICANN. The original top-level domains were seven three-letter domains – `.com`, `.net`, `.org`, `.int`, `.edu`, `.mil` and `.gov` – and the two-letter country-code domains (*eg* .us, .ca, .mx). Now there are hundreds of non-country top-level domains, such as `.aero`, `.biz`, `.info`, and, apparently, .wtf. Domain names (and subdomain names) can also contain unicode characters, so as to support national alphabets. Some top-level domains are *generic*, meaning anyone can apply for a subdomain although there may be qualifying criteria. Other top-level domains are *sponsored*, meaning the sponsoring organization determines who can be assigned a subdomain, and so the qualifying criteria can be a little more arbitrary.

ICANN still must approve all new top-level domains. Applications are accepted only during specific intervals; the application fee for the 2012 interval was US$185,000. The actual leasing of domain names to companies and individuals is done by organizations known as **domain registrars** who work under contract with ICANN.

The full tree of all DNS names and prefixes is divided administratively into **zones**: a zone is an independently managed subtree, minus any sub-subtrees that have been placed – by delegation – into their own zone. Each zone has its own root DNS name that is a suffix of every DNS name in the zone. For example, the `luc.edu` zone contains most of Loyola's DNS names, but `cs.luc.edu` has been spun off into its own zone. A zone cannot be the disjoint union of two subtrees; that is, `cs.luc.edu` and `math.luc.edu` must be two distinct zones, unless both remain part of their parent zone.

A zone can define DNS names more than one level deep. For example, the `luc.edu` zone can define records for the `luc.edu` name itself, for names with one additional level such as `www.luc.edu`, and for names with two additional levels such as `www.cs.luc.edu`. That said, it is common for each zone to handle only one additional level, and to create subzones for deeper levels.

Each zone has its own **authoritative nameservers** for the zone, which are charged with maintaining the records – known as **resource records**, or RRs – for that zone. Each zone must have at least two nameservers, for redundancy. IPv4 addresses are stored as so-called **A records**, for Address. Information about how to find sub-zones is stored as **NS records**, for Name Server. Additional resource-record types are discussed at *7.8.2 Other DNS Records*. An authoritative nameserver need not be part of the organization that manages the zone, and a single server can be the authoritative nameserver for multiple unrelated zones. For example, many domain registrars maintain single nameservers that handle DNS queries for all their domain customers who do not wish to set up their own nameservers.

The **root nameservers** handle the zone that is the root of the DNS tree; that is, that is represented by the DNS name that is the empty string. As of 2019, there are thirteen of them. The root nameservers contain only NS records, identifying the nameservers for all the immediate subzones. Each top-level domain is its own such subzone. The IP addresses of the root nameservers are widely distributed. Their DNS names (which are only of use if some DNS lookup mechanism is already present) are `a.root.servers.net` through `m.root-servers.net`. These names today correspond not to individual machines but to clusters of up to hundreds of servers.

We can now put together a first draft of a DNS lookup algorithm. To find the IP address of intronetworks.cs.luc.edu, a host first contacts a root nameserver (at a known address) to find the nameserver for the

edu zone; this involves the retrieval of an NS record. The edu nameserver is then queried to find the nameserver for the luc.edu zone, which in turn supplies the NS record giving the address of the cs.luc.edu zone. This last has an A record for the actual host. (This example is carried out in detail below.)

DNS Policing

It is sometimes suggested that if a site is engaged in illegal activity or copyright infringement, such as thepiratebay.se, its domain name should be seized. The problem with this strategy is that it is straightforward for users to set up "nonstandard" nameservers (for example the Gnu Name System, GNS) that continue to list the banned site.

This strategy has a defect in that it would send much too much traffic to the root nameservers. Instead, there exists a great number of local and semi-local "DNS servers" that we will call **resolvers** (though, confusingly, these are sometimes also known as "nameservers" or, more precisely, **non**-authoritative nameservers). A resolver is a host charged with looking up DNS names on behalf of a user or set of users, and returning corresponding addresses; for this reason they are sometimes called **recursive** nameservers (we return to recursive DNS lookups below).

Most ISPs and companies provide a resolver to handle the DNS needs of their customers and employees; we will refer to these as **site resolvers**. The IP addresses of these site resolvers is generally supplied via DHCP options (*7.10 Dynamic Host Configuration Protocol (DHCP)*); such resolvers are thus the default choice for DNS services.

Sometimes, however, users elect to use a DNS resolver not provided by their ISP or company; there are a number of **public DNS servers** (that is, resolvers) available. Such resolvers generally serve much larger areas. Common choices include OpenDNS, Google DNS (at 8.8.8.8), Cloudflare (at 1.1.1.1) and the Gnu Name System mentioned in the sidebar above, though there are many others. Searching for "public DNS server" turns up lists of them.

One advantage of using a public DNS server is that your local ISP can no longer track your DNS queries. On the other hand, the public server now *can*, so this becomes a matter of which you trust more (or less).

Some public DNS servers provide additional services, such as automatically filtering out domain names associated with security risks, or content inappropriate for young users. Sometimes there is a fee for this service.

A resolver uses the level-by-level algorithm above as a fallback, but also keeps a large **cache** of all the domain names (and other record types) that have been requested. A lifetime for each cache entry is provided by that entry's authoritative nameserver; these lifetimes are typically on the order of several days. Every DNS record has a TTL (time-to-live) value representing its maximum cache lifetime.

If I send a query to Loyola's site resolver for google.com, it is almost certainly in the cache. If I send a query for the misspelling googel.com, this may not be in the cache, but the .com top-level nameserver almost certainly *is* in the cache. From that nameserver my local resolver finds the nameserver for the googel.com zone, and from that finds the IP address of the googel.com host.

Applications almost always invoke DNS through library calls, such as Java's InetAddress.getByName(). The library forwards the query to the system-designated resolver (though browsers sometimes offer other DNS options; see *22.12.4 DNS over HTTPS*). We will return to DNS library calls in *11.1.3.3 The Client* and *12.6.1 The TCP Client*.

On unix-based systems, traditionally the IPv4 addresses of the local DNS resolvers were kept in a file `/etc/resolv.conf`. Typically this file was updated with the addresses of the current resolvers by DHCP (*7.10 Dynamic Host Configuration Protocol (DHCP)*), at the time the system received its IPv4 address. It is possible, though not common, to create special entries in `/etc/resolv.conf` so that queries about different domains are sent to different resolvers, or so that single-level hostnames have a domain name appended to them before lookup. On Windows, similar functionality can be achieved through settings on the DNS tab within the `Network Connections` applet.

Recent systems often run a small "stub" resolver locally (*eg* Linux's dnsmasq); such resolvers are sometimes also called DNS *forwarders*. The entry in `/etc/resolv.conf` is then an IPv4 address of `localhost` (sometimes 127.0.**1**.1 rather than 127.0.**0**.1). Such a stub resolver would, of course, still need access to the addresses of site or public resolvers; sometimes these addresses are provided by static configuration and sometimes by DHCP (*7.10 Dynamic Host Configuration Protocol (DHCP)*).

If a system running a stub resolver then runs internal virtual machines, it is usually possible to configure everything so that the virtual machines can be given an IP address of the host system as their DNS resolver. For example, often virtual machines are assigned IPv4 addresses on a private subnet and connect to the outside world using NAT (*7.7 Network Address Translation*). In such a setting, the virtual machines are given the IPv4 address of the host system interface that connects to the private subnet. It is then necessary to ensure that, on the host system, the local resolver accepts queries sent not only to the designated loopback address but also to the host system's private-subnet address. (Generally, local resolvers do *not* accept requests arriving from externally visible addresses.)

When someone submits a query for a nonexistent DNS name, the resolver is supposed to return an error message, technically known as **NXDOMAIN** (Non eXistent Domain). Some resolvers, however, have been configured to return the IP address of a designated web server; this is particularly common for ISP-provided site resolvers. Sometimes the associated web page is meant to be helpful, and sometimes it presents an offer to buy the domain name from a registrar. Either way, additional advertising may be displayed. Of course, this is completely useless to users who are trying to contact the domain name in question via a protocol (ssh, smtp) other than http.

At the DNS protocol layer, a DNS lookup query can be either **recursive** or **non-recursive**. If A sends to B a recursive query to resolve a given DNS name, then B takes over the job until it is finally able to return an answer to A. If The query is non-recursive, on the other hand, then if B is not an authoritative nameserver for the DNS name in question it returns either a failure notice or an NS record for the sub-zone that is the next step on the path. Almost all DNS requests from hosts to their site or public resolvers are recursive.

A basic DNS response consists of an ANSWER section, an AUTHORITY section and, optionally, an ADDITIONAL section. Generally a response to a lookup of a hostname contains an ANSWER section consisting of a single A record, representing a single IPv4 address. If a site has multiple servers that are entirely equivalent, however, it is possible to give them all the same hostname by configuring the authoritative nameserver to return, for the hostname in question, multiple A records listing, in turn, each of the server IPv4 addresses. This is sometimes known as **round-robin DNS**. It is a simple form of **load balancing**; see also *18.9.5 loadbalance31.py*. Consecutive queries to the nameserver should return the list of A records in different orders; ideally the same should also happen with consecutive queries to a local resolver that has the hostname in its cache. It is also common for a single server, with a single IPv4 address, to be identified by multiple DNS names; see the next section.

The response AUTHORITY section contains the DNS names of the authoritative nameservers responsible for the original DNS name in question. The ADDITIONAL section contains information the sender thinks is related; for example, this section often contains A records for the authoritative nameservers.

The Tor Project uses DNS-like names that end in ".onion". While these are not true DNS names in that they are not managed by the DNS hierarchy, they do work as such for Tor users; see **RFC 7686**. These names follow an unusual pattern: the next level of name is an 80-bit hash of the site's RSA public key (*22.9.1 RSA*), converted to sixteen ASCII bytes. For example, 3g2upl4pq6kufc4m.onion is apparently the Tor address for the search engine duckduckgo.com. Unlike DuckDuckGo, many sites try different RSA keys until they find one where at least some initial prefix of the hash looks more or less meaningful; for example, `nytimes2tsqtnxek.onion`. Facebook got very lucky in finding an RSA key whose corresponding Tor address is `facebookcorewwwi.onion` (though it is sometimes said that *fortune is infatuated with the wealthy*). This naming strategy is a form of **cryptographically generated addresses**; for another example see *8.6.4 Security and Neighbor Discovery*. The advantage of this naming strategy is that you don't need a certificate authority (*22.10.2.1 Certificate Authorities*) to verify a site's RSA key; the site name does it for you.

7.8.1 nslookup (and dig)

Let us trace a non-recursive lookup of `intronetworks.cs.luc.edu`, using the nslookup tool. The nslookup tool is time-honored, but also not completely up-to-date, so we also include examples using the dig utility (supposedly an acronym for "domain Internet groper"). Lines we type in `nslookup`'s interactive mode begin below with the prompt ">"; the shell prompt is "#". All `dig` commands are typed directly at the shell prompt.

The first step is to look up the IP address of the root nameserver `a.name-servers.net`. We can do this with a regular call to `nslookup` or `dig`, we can look this up in our nameserver's configuration files, or we can search for it on the Internet. The address is 198.41.0.4.

We now send our nonrecursive query to this address. The presence of the single hyphen in the `nslookup` command line below means that we want to use 198.41.0.4 as the nameserver rather than as the thing to be looked up; `dig` has places on the command line for both the nameserver (following the `@`) and the DNS name. For both commands, we use the `norecurse` option to send a nonrecursive query.

```
# nslookup -norecurse - 198.41.0.4
> intronetworks.cs.luc.edu
*** Can't find intronetworks.cs.luc.edu: No answer

# dig @198.41.0.4 intronetworks.cs.luc.edu +norecurse
```

These fail because by default `nslookup` and `dig` ask for an A record. What we want is an NS record: the name of the next zone down to ask. (We can tell the `dig` query failed to find an A record because there are zero records in the ANSWER section)

```
> set query=ns
> intronetworks.cs.luc.edu
edu   nameserver = a.edu-servers.net
...
a.edu-servers.net      internet address = 192.5.6.30

# dig @198.41.0.4 intronetworks.cs.luc.edu NS +norecurse
;; AUTHORITY SECTION:
edu.                    172800  IN      NS      b.edu-servers.net.
```

```
;; ADDITIONAL SECTION:
b.edu-servers.net.      172800  IN      A       192.33.14.30
```

The full responses in each case are a list of all nameservers for the .edu zone; we list only the first response above. Note that the full DNS name `intronetworks.cs.luc.edu` in the query here is *not* an exact match for the DNS name `.edu` in the resource record returned; the latter is a suffix of the former. Some newer resolvers send just the `.edu` part, to limit the user's privacy exposure.

We send the next NS query to a.edu-servers.net (which does appear in the full `dig` answer)

```
# nslookup -query=ns -norecurse - 192.5.6.30
> intronetworks.cs.luc.edu
...
Authoritative answers can be found from:
luc.edu nameserver = bcdnswt1.it.luc.edu.
bcdnswt1.it.luc.edu    internet address = 147.126.64.64

# dig @192.5.6.30 intronetworks.cs.luc.edu NS +norecurse
;; AUTHORITY SECTION:
luc.edu.                172800  IN      NS      bcdnswt1.it.luc.edu.
;; ADDITIONAL SECTION:
bcdnswt1.it.luc.edu.    172800  IN      A       147.126.64.64
```

(Again, we show only one of several `luc.edu` nameservers returned). We continue.

```
# nslookup -query=ns - -norecurse 147.126.64.64
> intronetworks.cs.luc.edu
...
Authoritative answers can be found from:
cs.luc.edu     nameserver = dns1.cs.luc.edu.
ns1.cs.luc.edu internet address = 147.126.2.44

# dig @147.126.64.64 intronetworks.cs.luc.edu NS +norecurse
;; AUTHORITY SECTION:
cs.luc.edu.             86400   IN      NS      ns1.cs.luc.edu.
;; ADDITIONAL SECTION:
ns1.cs.luc.edu.                 86400   IN      A       147.126.2.44
```

We now ask this last nameserver, for the `cs.luc.edu` zone, for the A record:

```
# nslookup -query=A -norecurse - 147.126.2.44
> intronetworks.cs.luc.edu
...
intronetworks.cs.luc.edu       canonical name = linode1.cs.luc.edu.
Name:  linode1.cs.luc.edu
Address: 162.216.18.28

# dig @147.126.2.44 intronetworks.cs.luc.edu A +norecurse
;; ANSWER SECTION:
intronetworks.cs.luc.edu. 300  IN      A       162.216.18.28
```

This is the first time we get an ANSWER section (versus the AUTHORITY section)

Here we get a **canonical name**, or CNAME, record. The server that hosts `intronetworks.cs.luc.edu` also hosts several other websites, with different names; for example, introcs.cs.luc.edu (at least as of 2015). This is known as **virtual hosting**. Rather than provide separate A records for each website name, DNS was set up to provide a `CNAME` record for each website name pointing to a single physical server name `linode1.cs.luc.edu`. Only one A record is then needed, for this server.

The `nslookup` request for an A record returned instead the CNAME record, together with the A record for that CNAME (this is the 162.216.18.28 above). This is done for convenience.

Note that the IPv4 address here, 162.216.18.28, is unrelated to Loyola's own IPv4 address block 147.126.0.0/16. The server `linode1.cs.luc.edu` is managed by an external provider; there is no connection between the DNS name hierarchy and the IP address hierarchy.

Finally, if we look up both `www.cs.luc.edu` and `cs.luc.edu`, we see they resolve to the same address. The use of `www` as a hostname for a domain's webserver is often considered unnecessary and old-fashioned; many users prefer the shorter, "naked" domain name, *eg* `cs.luc.edu`.

It might be tempting to create a CNAME record for the naked domain, `cs.luc.edu`, pointing to the full hostname `www.cs.luc.edu`. However, **RFC 1034** does not allow this:

> If a CNAME RR is present at a node, no other data should be present; this ensures that the data for a canonical name and its aliases cannot be different.

There are, however, several other DNS data records for `cs.luc.edu`: an NS record (above), a SOA, or Start of Authority, record containing various administrative data such as the expiration time, and an MX record, discussed in the following section. All this makes `www.cs.luc.edu` and `cs.luc.edu` ineluctably quite different. **RFC 1034** adds, "this rule also insures that a cached CNAME can be used without checking with an authoritative server for other RR types."

A better way to create a naked-domain record, at least from the perspective of DNS, is to give it its own A record. This does mean that, if the webserver address changes, there are now *two* DNS records that need to be updated, but this is manageable.

Recently ANAME records have been proposed to handle this issue; an ANAME is like a *limited* CNAME not subject to the **RFC 1034** restriction above. An ANAME record for a naked domain, pointing to another hostname rather than to address, is legal. See the Internet draft draft-hunt-dnsop-aname. Some large CDNs (*1.12.2 Content-Distribution Networks*) already implement similar DNS tweaks internally. This does not require end-user awareness; the user requests an A record and the ANAME is resolved at the CDN side.

Finally, there is also an argument, at least when HTTP (web) traffic is involved, that the www *not* be deprecated, and that the naked domain should instead be *redirected*, at the HTTP layer, to the full hostname. This simplifies some issues; for example, you now have only one website, rather than two. You no longer have to be concerned with the fact that HTTP cookies with and without the "www" are different. And some CDNs may not be able to handle website failover to another server if the naked domain is reached via an A record. But none of these are DNS issues.

7.8.2 Other DNS Records

Besides address lookups, DNS also supports a few other kinds of searches. The best known is probably **reverse DNS**, which takes an IP address and returns a name. This is slightly complicated by the fact that

one IP address may be associated with multiple DNS names. What DNS does in this case is to return the canonical name, or CNAME; a given address can have only one CNAME.

Given an IPv4 address, say 147.126.1.230, the idea is to reverse it and append to it the suffix `in-addr.arpa`.

```
230.1.126.147.in-addr.arpa
```

There is a DNS name hierarchy for names of this form, with zones and authoritative servers. If all this has been configured – which it often is not, especially for user workstations – a request for the `PTR` record corresponding to the above should return a DNS hostname. In the case above, the name luc.edu is returned (at least as of 2018).

`PTR` records are the only DNS records to have an entirely separate hierarchy; other DNS types fit into the "standard" hierarchy. For example, DNS also supports MX, or Mail eXchange, records, meant to map a domain name (which might not correspond to any hostname, and, if it does, is more likely to correspond to the name of a web server) to the hostname of a server that accepts email on behalf of the domain. In effect this allows an organization's domain name, *eg* `luc.edu`, to represent both a web server and, at a different IP address, an email server. MX records can even represent a *set* of IP addresses that accept email.

DNS has from the beginning supported TXT records, for arbitrary text strings. The email Sender Policy Framework (**RFC 7208**) was developed to make it harder for email senders to pretend to be a domain they are not; this involves inserting so-called SPF records as DNS TXT records (or as substrings of TXT records, if TXT is also being used for something else.

For example, a DNS query for TXT records of `google.com` (not gmail.com!) might yield (2018)

```
google.com      text = "docusign=05958488-4752-4ef2-95eb-aa7ba8a3bd0e"
google.com      text = "v=spf1 include:_spf.google.com ~all"
```

The SPF system is interested in only the second record; the "v=spf1" specifies the SPF version. This second record tells us to look up `_spf.google.com`. That lookup returns

```
text = "v=spf1 include:_netblocks.google.com include:_netblocks2.google.com␣
↪include:_netblocks3.google.com ~all"
```

Lookup of `_netblocks.google.com` then returns

```
text = "v=spf1 ip4:64.233.160.0/19 ip4:66.102.0.0/20 ip4:66.249.80.0/20␣
↪ip4:72.14.192.0/18 ip4:74.125.0.0/16 ip4:108.177.8.0/21 ip4:173.194.0.0/16␣
↪ip4:209.85.128.0/17 ip4:216.58.192.0/19 ip4:216.239.32.0/19 ~all"
```

If a host connects to an email server, and declares that it is delivering mail from someone at google.com, then the host's email list should occur in the list above, or in one of the other included lists. If it does not, there is a good chance the email represents spam.

Each DNS record (or "resource record") has a name (*eg* `cs.luc.edu`) and a type (*eg* `A` or `AAAA` or `NS` or `MX`). Given a name and type, the set of matching resource records is known as the **RRset** for that name and type (technically there is also a "class", but the class of all the DNS records we are interested in is `IN`, for Internet). When a nameserver responds to a DNS query, what is returned (in the ANSWER section) is always an entire RRset: the RRset of all resource records matching the name and type contained in the original query.

In many cases, RRsets have a single member, because many hosts have a single IPv4 address. However, this is not universal. We saw above the example of a single DNS name having multiple A records when round-robin DNS is used. A single DNS name might also have separate A records for the host's public and private IPv4 addresses. And perhaps most MX-record (Mail eXchange) RRsets have multiple entries, as organizations often prefer, for redundancy, to have more than one server that can receive email.

7.8.3 DNS Cache Poisoning

The classic DNS security failure, known as cache poisoning, occurs when an attacker has been able to convince a DNS resolver that the address of, say, www.example.com is something other than what it really is. A successful attack means the attacker can direct traffic meant for `www.example.com` to the attacker's own, malicious site.

The most basic cache-poisoning strategy is to send a stream of DNS reply packets to the resolver which declare that the IP address of www.example.com is the attacker's chosen IP address. The source IP address of these packets should be spoofed to be that of the example.com authoritative nameserver; such spoofing is relatively easy using UDP. Most of these reply packets will be ignored, but the hope is that one will arrive shortly after the resolver has sent a DNS request to the `example.com` authoritative nameserver, and interprets the spoofed reply packet as a legitimate reply.

To prevent this, DNS requests contain a 16-bit ID field; the DNS response must echo this back. The response must also come from the correct port. This leaves the attacker to guess 32 bits in all, but often the ID field (and even more often the port) can be guessed based on past history.

Another approach requires the attacker to wait for the target resolver to issue a legitimate request to the attacker's site, `attacker.com`. The attacker then piggybacks in the ADDITIONAL section of the reply message an A record for `example.com` pointing to the attacker's chosen bad IP address for this site. The hope is that the receiving resolver will place these A records from the ADDITIONAL section into its cache without verifying them further and without noticing they are completely unrelated. Once upon a time, such DNS resolver behavior was common.

Most newer DNS resolvers carefully validate the replies: the ID field must match, the source port must match, and any received DNS records in the ADDITIONAL section must match, at a minimum, the DNS zone of the request. Additionally, the request ID field and source port should be chosen pseudorandomly in a secure fashion. For additional vulnerabilities, see **RFC 3833**.

The central risk in cache poisoning is that a resolver can be tricked into supplying users with invalid DNS records. A closely related risk is that an attacker can achieve the same result by spoofing an authoritative nameserver. Both of these risks can be mitigated through the use of the DNS security extensions, known as **DNSSEC**. Because DNSSEC makes use of public-key signatures, we defer coverage to *22.12 DNSSEC*.

7.8.4 DNS and CDNs

DNS is often pressed into service by CDNs (*1.12.2 Content-Distribution Networks*) to identify their closest "edge" server to a given user. Typically this involves the use of **geoDNS**, a slightly nonstandard variation of DNS. When a DNS query comes in to one of the CDN's authoritative nameservers, that server

1. looks up the approximate location of the client (*10.4.4 IP Geolocation*)
2. determines the closest edge server to that location

3. replies with the IP address of that closest edge server

This works reasonably well most of the time. However, the requesting client is essentially never the end user; rather, it is the DNS resolver being used by the user. Typically such resolvers are the site resolvers provided by the user's ISP or organization, and are physically quite close to the user; in this case, the edge server identified above will be close to the user as well. However, when a user has chosen a (likely remote) public DNS resolver, as above, the IP address returned for the CDN edge server will be close to the DNS resolver but likely far from optimal for the end user.

One solution to this last problem is addressed by **RFC 7871**, which allows DNS resolvers to include the IP address of the client in the request sent to the authoritative nameserver. For privacy reasons, usually only a prefix of the user's IP address is included, perhaps /24. Even so, user's privacy is at least partly compromised. For this reason, **RFC 7871** recommends that the feature be disabled by default, and only enabled after careful analysis of the tradeoffs.

A user who is concerned about the privacy issue can – in theory – configure their own DNS software to include this **RFC 7871** option with a zero-length prefix of the user's IP address, which conveys no address information. The user's resolver will then not change this to a longer prefix.

Use of this option also means that the DNS resolver receiving a user query about a given hostname can no longer simply return a cached answer from a previous lookup of the hostname. Instead, the resolver needs to cache separately each ⟨hostname,prefix⟩ pair it handles, where the prefix is the prefix of the user's IP address forwarded to the authoritative nameserver. This has the potential to increase the cache size by several orders of magnitude, which may thereby enable some cache-overflow attacks.

7.9 Address Resolution Protocol: ARP

If a host or router A finds that the destination IP address D = D_{IP} matches the network address of one of its interfaces, it is to deliver the packet via the LAN (probably Ethernet). This means looking up the LAN address (MAC address) D_{LAN} corresponding to D_{IP}. How does it do this?

One approach would be via a special server, but the spirit of early IPv4 development was to avoid such servers, for both cost and reliability issues. Instead, the **Address Resolution Protocol** (ARP) is used. This is our first protocol that takes advantage of the existence of LAN-level broadcast; on LANs without physical broadcast (such as ATM), some other mechanism (usually involving a server) must be used.

The basic idea of ARP is that the host A sends out a broadcast ARP query or "who-has D_{IP}?" request, which includes A's own IPv4 and LAN addresses. All hosts on the LAN receive this message. The host for whom the message is intended, D, will recognize that it should reply, and will return an ARP reply or "is-at" message containing D_{LAN}. Because the original request contained A_{LAN}, D's response can be sent directly to A, that is, unicast.

A broadcasts "who-has D"
D replies to A, unicast, and includes its LAN address

Additionally, all hosts maintain an **ARP cache**, consisting of ⟨IPv4,LAN⟩ address pairs for other hosts on the network. After the exchange above, A has $\langle D_{IP}, D_{LAN} \rangle$ in its table; anticipating that A will soon send it a packet to which it needs to respond, D also puts $\langle A_{IP}, A_{LAN} \rangle$ into its cache.

ARP-cache entries eventually expire. The timeout interval used to be on the order of 10 minutes, but Linux systems now use a much smaller timeout (~30 seconds observed in 2012). Somewhere along the line, and probably related to this shortened timeout interval, repeat ARP queries about a *timed-out* entry are first sent **unicast**, not broadcast, to the previous Ethernet address on record. This cuts down on the total amount of broadcast traffic; LAN broadcasts are, of course, still needed for new hosts. The ARP cache on a Linux system can be examined with the command `ip -s neigh`; the corresponding windows command is `arp -a`.

The above protocol is sufficient, but there is one further point. When A sends its broadcast "who-has D?" ARP query, all other hosts C check their own cache for an entry for **A**. If there *is* such an entry (that is, if A_{IP} is found there), then the value for A_{LAN} is updated with the value taken from the ARP message; if there is no pre-existing entry then no action is taken. This update process serves to avoid stale ARP-cache entries, which can arise is if a host has had its Ethernet interface replaced. (USB Ethernet interfaces, in particular, can be replaced very quickly.)

ARP is quite an efficient mechanism for bridging the gap between IPv4 and LAN addresses. Nodes generally find out neighboring IPv4 addresses through higher-level protocols, and ARP then quickly fills in the missing LAN address. However, in some Software-Defined Networking (*2.8 Software-Defined Networking*) environments, the LAN switches and/or the LAN controller may have knowledge about IPv4/LAN address correspondences, potentially making ARP superfluous. The LAN (Ethernet) switching network might in principle even know exactly how to route *via the LAN* to a given IPv4 address, potentially even making LAN addresses unnecessary. At such a point, ARP may become an inconvenience. For an example of a situation in which it is necessary to work around ARP, see *18.9.5 loadbalance31.py*.

7.9.1 ARP Finer Points

Most hosts today implement **self-ARP**, or **gratuitous ARP**, on startup (or wakeup): when station A starts up it sends out an ARP query *for itself*: "who-has A?". Two things are gained from this: first, all stations that had A in their cache are now updated with A's most current A_{LAN} address, in case there was a change, and second, if an answer is received, then presumably some other host on the network has the same IPv4 address as A.

Self-ARP is thus the traditional IPv4 mechanism for **duplicate address detection**. Unfortunately, it does not always work as well as might be hoped; often only a single self-ARP query is sent, and if a reply is received then frequently the only response is to log an error message; the host may even continue using the duplicate address! If the duplicate address was received via DHCP, below, then the host is supposed to notify its DHCP server of the error and request a different IPv4 address.

RFC 5227 has defined an improved mechanism known as **Address Conflict Detection**, or ACD. A host using ACD sends out three ARP queries for its new IPv4 address, spaced over a few seconds and leaving the ARP field for the sender's IPv4 address filled with zeroes. This last step means that any other host with that IPv4 address in its cache will ignore the packet, rather than update its cache. If the original host receives no replies, it then sends out two more ARP queries for its new address, this time with the ARP field for the sender's IPv4 address filled in with the new address; this is the stage at which other hosts on the network will make any necessary cache updates. Finally, ACD requires that hosts that do detect a duplicate address

must discontinue using it.

It is also possible for other stations to answer an ARP query on behalf of the actual destination D; this is called **proxy ARP**. An early common scenario for this was when host C on a LAN had a modem connected to a serial port. In theory a host D dialing in to this modem should be on a different subnet, but that requires allocation of a new subnet. Instead, many sites chose a simpler arrangement. A host that dialed in to C's serial port might be assigned IP address D_{IP}, from the same subnet as C. C would be configured to route packets to D; that is, packets arriving from the serial line would be forwarded to the LAN interface, and packets sent to C_{LAN} addressed to D_{IP} would be forwarded to D. But we also have to handle ARP, and as D is not actually on the LAN it will not receive broadcast ARP queries. Instead, C would be configured to answer on behalf of D, replying with $\langle D_{IP}, \mathbf{C}_{LAN} \rangle$. This generally worked quite well.

Proxy ARP is also used in Mobile IP, for the so-called "home agent" to intercept traffic addressed to the "home address" of a mobile device and then forward it (*eg* via tunneling) to that device. See *7.13 Mobile IP*.

One delicate aspect of the ARP protocol is that stations are required to respond to a **broadcast** query. In the absence of proxies this theoretically should not create problems: there should be only one respondent. However, there were anecdotes from the Elder Days of networking when a broadcast ARP query would trigger an avalanche of responses. The protocol-design moral here is that determining who is to respond to a broadcast message should be done with great care. (**RFC 1122** section 3.2.2 addresses this same point in the context of responding to broadcast ICMP messages.)

ARP-query implementations also need to include a timeout and some queues, so that queries can be resent if lost and so that a burst of packets does not lead to a burst of queries. A naive ARP algorithm without these might be:

> To send a packet to destination D_{IP}, see if D_{IP} is in the ARP cache. If it is, address the packet to D_{LAN}; if not, send an ARP query for D

To see the problem with this approach, imagine that a 32 kB packet arrives at the IP layer, to be sent over Ethernet. It will be fragmented into 22 fragments (assuming an Ethernet MTU of 1500 bytes), all sent at once. The naive algorithm above will likely send an ARP query for *each* of these. What we need instead is something like the following:

> To send a packet to destination D_{IP}:
> If D_{IP} is in the ARP cache, send to D_{LAN} and return
> If not, see if an ARP query for D_{IP} is pending.
> If it is, put the current packet in a queue for D.
> If there is no pending ARP query for D_{IP}, start one,
> again putting the current packet in the (new) queue for D

We also need:

> If an ARP query for some C_{IP} times out, resend it (up to a point)
> If an ARP query for C_{IP} is answered, send off any packets in C's queue

7.9.2 ARP Security

Suppose A wants to log in to secure server S, using a password. How can B (for Bad) impersonate S?

Here is an ARP-based strategy, sometimes known as **ARP Spoofing**. First, B makes sure the real S is down, either by waiting until scheduled downtime or by launching a denial-of-service attack against S.

When A tries to connect, it will begin with an ARP "who-has S?". All B has to do is answer, "S is-at B". There is a trivial way to do this: B simply needs to set its own IP address to that of S.

A will connect, and may be convinced to give its password to B. B now simply responds with something plausible like "backup in progress; try later", and meanwhile use A's credentials against the real S.

This works even if the communications channel A uses is encrypted! If A is using the SSH protocol (*22.10.1 SSH*), then A will get a message that the other side's key has changed (B will present its own SSH key, not S's). Unfortunately, many users (and even some IT departments) do not recognize this as a serious problem. Some organizations – especially schools and universities – use personal workstations with "frozen" configuration, so that the filesystem is reset to its original state on every reboot. Such systems may be resistant to viruses, but in these environments the user at A will always get a message to the effect that S's credentials are not known.

7.9.3 ARP Failover

Suppose you have two front-line servers, A and B (B for Backup), and you want B to be able to step in if A freezes. There are a number of ways of achieving this, but one of the simplest is known as **ARP Failover**. First, we set $A_{IP} = B_{IP}$, but for the time being B does not use the network so this duplication is not a problem. Then, once B gets the message that A is down, it sends out an ARP query for A_{IP}, including B_{LAN} as the source LAN address. The gateway router, which previously would have had $\langle A_{IP}, A_{LAN} \rangle$ in its ARP cache, updates this to $\langle A_{IP}, \mathbf{B}_{LAN} \rangle$, and packets that had formerly been sent to A will now go to B. As long as B is trafficking in stateless operations (*eg* html), B can pick up right where A left off.

7.9.4 Detecting Sniffers

Finally, there is an interesting use of ARP to detect Ethernet password sniffers (generally not quite the issue it once was, due to encryption and switching). To find out if a particular host A is in promiscuous mode, send an ARP "who-has A?" query. Address it not to the broadcast Ethernet address, though, but to some nonexistent Ethernet address.

If promiscuous mode is off, A's network interface will ignore the packet. But if promiscuous mode is on, A's network interface will pass the ARP request to A itself, which is likely then to answer it.

Alas, Linux kernels reject at the ARP-software level ARP queries to physical Ethernet addresses other than their own. However, they do respond to faked Ethernet multicast addresses, such as ff:ff:ff:00:00:00 or ff:ff:ff:ff:ff:fe.

7.9.5 ARP and multihomed hosts

If host A has two interfaces `iface1` and `iface2` *on the same LAN*, with respective IP addresses A_1 and A_2, then it is common for the two to be used interchangeably. Traffic addressed to A_1 may be received via `iface2` and vice-versa, and traffic *from* A_1 may be sent via `iface2`. In *7.2.1 Multihomed hosts* this is described as the **weak end-system** model; the idea is that we should think of the IP addresses A_1 and A_2 as bound to A rather than to their respective interfaces.

In support of this model, ARP can usually be configured (in fact this is often the default) so that ARP requests for either IP address and received by either interface may be answered with either physical address. Usually all requests are answered with the physical address of the preferred (*ie* faster) interface.

As an example, suppose A has an Ethernet interface `eth0` with IP address 10.0.0.2 and a faster Wi-Fi interface `wlan0` with IP address 10.0.0.3 (although Wi-Fi interfaces are *not* always faster). In this setting, an ARP request "who-has 10.0.0.2" would be answered with `wlan0`'s physical address, and so all traffic to A, to either IP address, would arrive via `wlan0`. The `eth0` interface would go essentially unused. Similarly, though not due to ARP, traffic sent by A with source address 10.0.0.2 might depart via `wlan0`.

This situation is on Linux systems adjusted by changing `arp_ignore` and `arp_announce` in `/proc/sys/net/ipv4/conf/all`.

7.10 Dynamic Host Configuration Protocol (DHCP)

DHCP is the most common mechanism by which hosts are assigned their IPv4 addresses. DHCP started as a protocol known as Reverse ARP (RARP), which evolved into BOOTP and then into its present form. It is documented in **RFC 2131**. Recall that ARP is based on the idea of someone broadcasting an ARP query for a host, containing the host's IPv4 address, and the host answering it with its LAN address. DHCP involves a host, at startup, broadcasting a query containing its *own* LAN address, and having a server reply telling the host what IPv4 address is assigned to it, hence the "Reverse ARP" name.

The DHCP response message is also likely to carry, piggybacked onto it, several other essential startup options. Unlike the IPv4 address, these additional network parameters usually do not depend on the specific host that has sent the DHCP query; they are likely constant for the subnet or even the site. In all, a typical DHCP message includes the following:

- IPv4 address
- subnet mask
- default router
- DNS Server

These four items are a standard **minimal network configuration**; in practical terms, hosts cannot function properly without them. Most DHCP implementations support the piggybacking of the latter three above, and a wide variety of other configuration values, onto the server responses.

Default Routers and DHCP

If you lose your default router, you cannot communicate. Here is something that used to happen to me, courtesy of DHCP:

1. I am connected to the Internet via Ethernet, and my default router is via my Ethernet interface
2. I connect to my institution's wireless network.
3. Their DHCP server sends me a new default router on the wireless network. However, this default router will only allow access to a tiny private network, because I have neglected to complete the "Wi-Fi network registration" process.

4. I therefore disconnect from the wireless network, and my wireless-interface default router goes away. However, my system does not automatically revert to my Ethernet default-router entry; DHCP does not work that way. As a result, I will have no router at all until the next scheduled DHCP lease renegotiation, and must fix things manually.

The DHCP server has a range of IPv4 addresses to hand out, and maintains a database of which IPv4 address has been assigned to which LAN address. Reservations can either be permanent or dynamic; if the latter, hosts typically renew their DHCP reservation periodically (typically one to several times a day).

7.10.1 NAT, DHCP and the Small Office

If you have a large network, with multiple subnets, a certain amount of manual configuration is inevitable. What about, however, a home or small office, with a single line from an ISP? A combination of NAT (*7.7 Network Address Translation*) and DHCP has made **autoconfiguration** close to a reality.

The typical home/small-office "router" is in fact a NAT router (*7.7 Network Address Translation*) coupled with an Ethernet switch, and usually also coupled with a Wi-Fi access point and a DHCP server. In this section, we will use the term "NAT router" to refer to this whole package. One specially designated port, the **external** port, connects to the ISP's line, and uses DHCP as a client to obtain an IPv4 address for that port. The other, **internal**, ports are connected together by an Ethernet switch; these ports as a group are connected to the external port using NAT translation. If wireless is supported, the wireless side is connected directly to the internal ports.

Isolated from the Internet, the internal ports can thus be assigned an arbitrary non-public IPv4 address block, *eg* 192.168.0.0/24. The NAT router typically contains a DCHP server, usually enabled by default, that will hand out IPv4 addresses to everything connecting from the internal side.

Generally this works seamlessly. However, if a second NAT router is also connected to the network (sometimes attempted to extend Wi-Fi range, in lieu of a commercial Wi-Fi repeater), one then has two operating DHCP servers on the same subnet. This often results in chaos, though is easily fixed by disabling one of the DHCP servers.

While omnipresent DHCP servers have made IPv4 autoconfiguration work "out of the box" in many cases, in the era in which IPv4 was designed the need for such servers would have been seen as a significant drawback in terms of expense and reliability. IPv6 has an autoconfiguration strategy (*8.7.2 Stateless Autoconfiguration (SLAAC)*) that does not require DHCP, though DHCPv6 may well end up displacing it.

7.10.2 DHCP and Routers

It is often desired, for larger sites, to have only one or two DHCP servers, but to have them support multiple subnets. Classical DHCP relies on broadcast, which isn't forwarded by routers, and even if it were, the DHCP server would have no way of knowing on what subnet the host in question was actually located.

This is generally addressed by **DHCP Relay** (sometimes still known by the older name BOOTP Relay). The router (or, sometimes, some other node on the subnet) receives the DHCP broadcast message from a host, and notes the subnet address of the arrival interface. The router then relays the DHCP request, together with this subnet address, to the designated DHCP Server; this relayed message is sent directly (unicast), not

broadcast. Because the subnet address is included, the DHCP server can figure out the correct IPv4 address to assign.

This feature has to be specially enabled on the router.

7.11 Internet Control Message Protocol

The Internet Control Message Protocol, or ICMP, is a protocol for sending IP-layer error and status messages; it is defined in RFC 792. ICMP is, like IP, **host-to-host**, and so they are never delivered to a specific port, even if they are sent in response to an error related to something sent from that port. In other words, individual UDP and TCP connections do not receive ICMP messages, even when it would be helpful to get them.

ICMP messages are identified by an 8-bit **type** field, followed by an 8-bit subtype, or **code**. Here are the more common ICMP types, with subtypes listed in the description.

Type	Description
Echo Request	`ping` queries
Echo Reply	`ping` responses
Destination Unreachable	Destination **network** unreachable
	Destination **host** unreachable
	Destination **port** unreachable
	Fragmentation required but DF flag set
	Network administratively prohibited
Source Quench	Congestion control
Redirect Message	Redirect datagram for the **network**
	Redirect datagram for the **host**
	Redirect for TOS and network
	Redirect for TOS and host
Router Solicitation	Router discovery/selection/solicitation
Time Exceeded	TTL expired in transit
	Fragment reassembly time exceeded
Bad IP Header or Parameter	Pointer indicates the error
	Missing a required option
	Bad length
Timestamp Timestamp Reply	Like `ping`, but requesting a timestamp from the destination

The Echo and Timestamp formats are *queries*, sent by one host to another. Most of the others are all *error messages*, sent by a router to the sender of the offending packet. Error-message formats contain the IP header and next 8 bytes of the packet in question; the 8 bytes will contain the TCP or UDP port numbers. Redirect and Router Solicitation messages are informational, but follow the error-message format. Query formats contain a 16-bit *Query Identifier*, assigned by the query sender and echoed back by the query responder.

`ping` Packet Size

The author once had to diagnose a problem where pings were *almost* 100% successful, and yet file transfers failed immediately; this could have been the result of either a network fault or a file-transfer application fault. The problem turned out to be a failed network device with a very high bit-error rate: 1500-byte file-transfer packets were frequently corrupted, but ping packets, with a default size of 32-64 bytes, were mostly unaffected. If the bit-error rate is such that 1500-byte packets have a 50% success rate, 50-byte packets can be expected to have a 98% ($\simeq 0.5^{1/30}$) success rate. Setting the ping packet size to a larger value made it immediately clear that the network, and not the file-transfer application, was at fault.

ICMP is perhaps best known for Echo Request/Reply, on which the `ping` tool (*1.14 Some Useful Utilities*) is based. Ping remains very useful for network troubleshooting: if you can ping a host, then the network is reachable, and any problems are higher up the protocol chain. Unfortunately, ping replies are often blocked by many firewalls, on the theory that revealing even the existence of computers is a security risk. While this may sometimes be an appropriate decision, it does significantly impair the utility of ping.

Ping can be asked to include IP timestamps (*7.1 The IPv4 Header*) on Linux systems with the `-T` option, and on Windows with `-s`.

Source Quench was used to signal that congestion has been encountered. A router that drops a packet due to congestion experience was encouraged to send ICMP Source Quench to the originating host. Generally the TCP layer would handle these appropriately (by reducing the overall sending rate), but UDP applications never receive them. ICMP Source Quench did not quite work out as intended, and was formally deprecated by **RFC 6633**. (Routers can inform TCP connections of impending congestion by using the ECN bits.)

The Destination Unreachable type has a large number of subtypes:

- **Network unreachable**: some router had no entry for forwarding the packet, and no default route
- **Host unreachable**: the packet reached a router that was on the same LAN as the host, but the host failed to respond to ARP queries
- **Port unreachable**: the packet was sent to a UDP port on a given host, but that port was not open. TCP, on the other hand, deals with this situation by replying to the connecting endpoint with a `reset` packet. Unfortunately, the UDP Port Unreachable message is sent to the host, not to the application on that host that sent the undeliverable packet, and so is close to useless as a practical way for applications to be informed when packets cannot be delivered.
- **Fragmentation required but DF flag set**: a packet arrived at a router and was too big to be forwarded without fragmentation. However, the Don't Fragment bit in the IPv4 header was set, forbidding fragmentation.
- **Administratively Prohibited**: this is sent by a router that knows it can reach the network in question, but has configureintro to drop the packet and send back Administratively Prohibited messages. A router can also be configured to **blackhole** messages: to drop the packet and send back nothing.

In *12.13 Path MTU Discovery* we will see how TCP uses the ICMP message **Fragmentation required but DF flag set** as part of **Path MTU Discovery**, the process of finding the largest packet that can be sent *to a specific destination* without fragmentation. The basic idea is that we set the DF bit on some of the packets we send; if we get back this message, that packet was too big.

Some sites and firewalls block ICMP packets in addition to Echo Request/Reply, and for some messages one can get away with this with relatively few consequences. However, blocking **Fragmentation required but DF flag set** has the potential to severely affect TCP connections, depending on how Path MTU Discovery is

implemented, and thus is not recommended. If ICMP filtering is contemplated, it is best to base block/allow decisions on the ICMP type, or even on the type and code. For example, most firewalls support rule sets of the form "allow ICMP destination-unreachable; block all other ICMP".

The **Timestamp** option works something like Echo Request/Reply, but the receiver includes its own local timestamp for the arrival time, with millisecond accuracy. See also the IP Timestamp option, *7.1 The IPv4 Header*, which appears to be more frequently used.

The type/code message format makes it easy to add new ICMP types. Over the years, a significant number of additional such types have been defined; a complete list is maintained by the IANA. Several of these later ICMP types were seldom used and eventually deprecated, many by **RFC 6918**.

ICMP packets are usually forwarded correctly through NAT routers, though due to the absence of port numbers the router must do a little more work. **RFC 3022** and **RFC 5508** address this. For ICMP queries, like ping, the ICMP Query Identifier field can be used to recognize the returning response. ICMP error messages are a little trickier, because there is no direct connection between the inbound error message and any of the previous outbound non-ICMP packets that triggered the response. However, the headers of the packet that triggered the ICMP error message are embedded in the body of the ICMP message. The NAT router can look at those embedded headers to determine how to forward the ICMP message (the NAT router must also rewrite the addresses of those embedded headers).

7.11.1 Traceroute and Time Exceeded

The traceroute program uses ICMP Time Exceeded messages. A packet is sent to the destination (often UDP to an unused port), with the TTL set to 1. The first router the packet reaches decrements the TTL to 0, drops it, and returns an ICMP Time Exceeded message. The sender now knows the first router on the chain. The second packet is sent with TTL set to 2, and the second router on the path will be the one to return ICMP Time Exceeded. This continues until finally the remote host returns something, likely ICMP Port Unreachable.

For an example of traceroute output, see *1.14 Some Useful Utilities*. In that example, the three traceroute probes for the Nth router are sometimes answered by two or even three different routers; this suggests routers configured to work in parallel rather than route changes.

Many routers no longer respond with ICMP Time Exceeded messages when they drop packets. For the distance value corresponding to such a router, traceroute reports $\star\star\star$.

Traceroute assumes the path does not change. This is not always the case, although in practice it is seldom an issue.

Route Efficiency

Once upon a time (~2001), traceroute showed that traffic from my home to the office, both in the Chicago area, went through the MAE-EAST Internet exchange point, outside of Washington DC. That inefficient route was later fixed. A situation like this is typically caused by two higher-level providers who did not negotiate sufficient Internet exchange points.

Traceroute to a nonexistent site works up to the point when the packet reaches the Internet "backbone": the first router which does not have a default route. At that point the packet is not routed further (and an ICMP

Destination Network Unreachable should be returned).

Traceroute also interacts somewhat oddly with routers using MPLS (see *20.12 Multi-Protocol Label Switching (MPLS)*). Such routers – most likely large-ISP internal routers – may continue to forward the ICMP Time Exceeded message on further towards its destination before returning it to the sender. As a result, the round-trip time measurements reported may be quite a bit larger than they should be.

7.11.2 Redirects

Most non-router hosts start up with an IPv4 forwarding table consisting of a single (default) router, discovered along with their IPv4 address through DHCP. ICMP Redirect messages help hosts learn of other useful routers. Here is a classic example:

A is configured so that its default router is R1. It addresses a packet to B, and sends it to R1. R1 receives the packet, and forwards it to R2. However, R1 also notices that R2 and A are on the same network, and so A could have sent the packet to R2 directly. So R1 sends an appropriate ICMP redirect message to A ("Redirect Datagram for the Network"), and A adds a route to B via R2 to its own forwarding table.

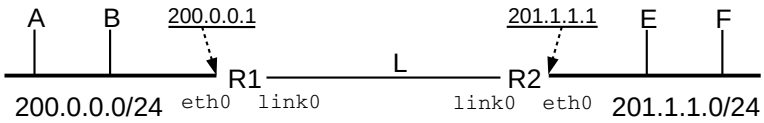
7.11.3 Router Solicitation

These ICMP messages are used by some router protocols to identify immediate neighbors. When we look at routing-update algorithms, *9 Routing-Update Algorithms*, these are where the process starts.

7.12 Unnumbered Interfaces

We mentioned in *1.10 IP - Internet Protocol* and *7.2 Interfaces* that some devices allow the use of point-to-point IP links without assigning IP addresses to the interfaces at the ends of the link. Such IP interfaces are referred to as **unnumbered**; they generally make sense only on routers. It is a firm requirement that the node (*ie* router) at each endpoint of such a link has at least one other interface that *does* have an IP address; otherwise, the node in question would be anonymous, and could not participate in the router-to-router protocols of *9 Routing-Update Algorithms*.

The diagram below shows a link L joining routers R1 and R2, which are connected to subnets 200.0.0.0/24 and 201.1.1.0/24 respectively. The endpoint interfaces of L, both labeled `link0`, are unnumbered.



Two LANs joined by an unnumbered link L

The endpoints of L could always be assigned private IPv4 addresses (*7.3 Special Addresses*), such as 10.0.0.1 and 10.0.0.2. To do this we would need to create a subnet; because the host bits cannot be all 0's or all 1's, the minimum subnet size is four (*eg* 10.0.0.0/30). Furthermore, the routing protocols to be introduced in *9 Routing-Update Algorithms* will distribute information about the subnet throughout the organization or "routing domain", meaning care must be taken to ensure that each link's subnet is unique. Use of unnumbered links avoids this.

If R1 were to *originate* a packet to be sent to (or forwarded via) R2, the standard strategy is for it to treat its `link0` interface as if it shared the IP address of its Ethernet interface `eth0`, that is, 200.0.0.1; R2 would do likewise. This still leaves R1 and R2 violating the IP local-delivery rule of *7.5 The Classless IP Delivery Algorithm*; R1 is expected to deliver packets via local delivery to 201.1.1.1 but has no interface that is assigned an IP address on the destination subnet 201.1.1.0/24. The necessary dispensation, however, is granted by **RFC 1812**. All that is necessary by way of configuration is that R1 be told R2 is a directly connected neighbor reachable via its `link0` interface. On Linux systems this might be done with the `ip route` command on R1 as follows:

ip route

The Linux `ip route` command illustrated here was tested on a virtual point-to-point link created with `ssh` and `pppd`; the link interface name was in fact `ppp0`. While the command appeared to work as advertised, it was only possible to create the link if endpoint IP addresses were assigned at the time of creation; these were then removed with `ip route del` and then re-assigned with the command shown here.

```
ip route add 201.1.1.1 dev link0
```

Because L is a point-to-point link, there is no destination LAN address and thus no ARP query.

7.13 Mobile IP

In the original IPv4 model, there was a strong if implicit assumption that each IP host would stay put. One role of an IPv4 address is simply as a unique endpoint identifier, but another role is as a **locator**: some prefix of the address (*eg* the network part, in the class-A/B/C strategy, or the provider prefix) represents something about where the host is physically located. Thus, if a host moves far enough, it may need a new address.

When laptops are moved from site to site, it is common for them to receive a new IP address at each location, *eg* via DHCP as the laptop connects to the local Wi-Fi. But what if we wish to support devices like smartphones that may remain active and communicating while moving for thousands of miles? Changing IP addresses requires changing TCP connections; life (and application development) might be simpler if a device had a single, unchanging IP address.

One option, commonly used with smartphones connected to some so-called "3G" networks, is to treat the phone's data network as a giant wireless LAN. The phone's IP address need not change as it moves within this LAN, and it is up to the phone provider to figure out how to manage LAN-level routing, much as is done in *3.7.4.3 Roaming*.

But **Mobile IP** is another option, documented in **RFC 5944**. In this scheme, a mobile host has a permanent **home address** and, while roaming about, will also have a temporary **care-of address**, which changes from place to place. The care-of address might be, for example, an IP address assigned by a local Wi-Fi network, and which in the absence of Mobile IP would be *the* IP address for the mobile host. (This kind of care-of address is known as "co-located"; the care-of address can also be associated with some other device – known as a **foreign agent** – in the vicinity of the mobile host.) The goal of Mobile IP is to make sure that the mobile host is always reachable via its home address.

To maintain connectivity to the home address, a Mobile IP host needs to have a **home agent** back on the home network; the job of the home agent is to maintain an IP tunnel that always connects to the device's current care-of address. Packets arriving at the home network addressed to the home address will be forwarded to the mobile device over this tunnel by the home agent. Similarly, if the mobile device wishes to send packets *from* its home address – that is, with the home address as IP source address – it can use the tunnel to forward the packet to the home agent.

The home agent may use proxy ARP (*7.9.1 ARP Finer Points*) to declare itself to be the appropriate destination on the home LAN for packets addressed to the home (IP) address; it is then straightforward for the home agent to forward the packets.

An **agent discovery** process is used for the mobile host to decide whether it is mobile or not; if it is, it then needs to notify its home agent of its current care-of address.

7.13.1 IP-in-IP Encapsulation

There are several forms of packet encapsulation that can be used for Mobile IP tunneling, but the default one is IP-in-IP encapsulation, defined in **RFC 2003**. In this process, the entire original IP packet (with header addressed to the home address) is used as data for a new IP packet, with a new IP header (the "outer" header) addressed to the care-of address.

Original packet

Encapsulated packet with outer header

A value of 4 in the outer-IP-header `Protocol` field indicates that IPv4-in-IPv4 tunneling is being used, so the receiver knows to forward the packet on using the information in the inner header. The MTU of the tunnel will be the original MTU of the path to the care-of address, minus the size of the outer header. A very

similar mechanism is used for IPv6-in-IPv4 encapsulation (that is, with IPv6 in the inner packet), except that the outer IPv4 `Protocol` field value is now 41. See *8.13 IPv6 Connectivity via Tunneling*.

IP-in-IP encapsulation presents some difficulties for NAT routers. If two hosts A and B behind a NAT router send out encapsulated packets, the packets may differ only in the source IP address. The NAT router, upon receiving responses, doesn't know whether to forward them to A or to B. One partial solution is for the NAT router to support only one inside host sending encapsulated packets. If the NAT router knew that encapsulation was being used for Mobile IP, it might look at the home address in the inner header to determine the correct home agent to which to deliver the packet, but this is a big assumption. A fuller solution is outlined in **RFC 3519**.

7.14 Epilog

At this point we have concluded the basic mechanics of IPv4. Still to come is a discussion of how IP routers build their forwarding tables. This turns out to be a complex topic, divided into routing within single organizations and ISPs – *9 Routing-Update Algorithms* – and routing between organizations – *10 Large-Scale IP Routing*.

But before that, in the next chapter, we compare IPv4 with IPv6, now twenty years old but still seeing limited adoption. The biggest issue fixed by IPv6 is IPv4's lack of address space, but there are also several other less dramatic improvements.

7.15 Exercises

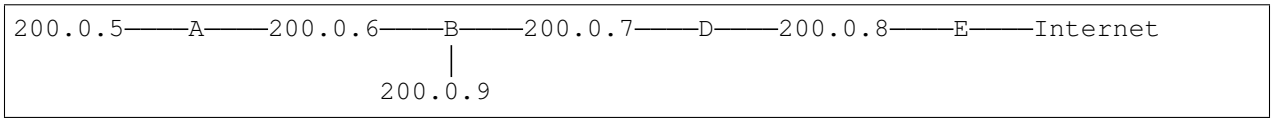
Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 6.5 is distinct, for example, from exercises 6.0 and 7.0. Exercises marked with a ♢ have solutions or hints at 24.7 Solutions for IPv4.

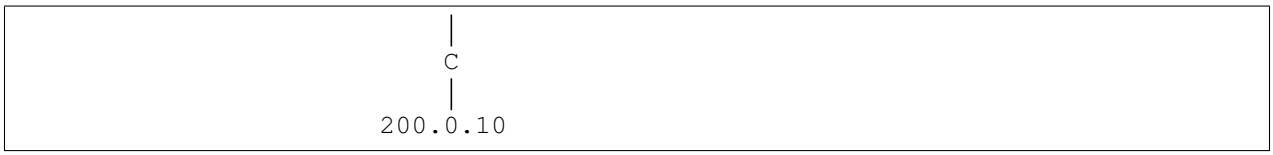
1.0. Suppose an Ethernet packet represents a TCP acknowledgment; that is, the packet contains an IPv4 header and a 20-byte TCP header but nothing else. Is the IPv4 packet here smaller than the Ethernet minimum-packet size, and, if so, by how much?

2.0. How can a receiving host tell if an arriving IPv4 packet is unfragmented? Hint: such a packet will be both the "first fragment" and the "last fragment"; how are these two states marked in the IPv4 header?

3.0. How long will it take the IDENT field of the IPv4 header to wrap around, if the sender host A sends a stream of packets to host B as fast as possible? Assume the packet size is 1500 bytes and the bandwidth is 600 Mbps.

4.0. The following diagram has routers A, B, C, D and E; E is the "border router" connecting the site to the Internet. All router-to-router connections are via Ethernet-LAN /24 subnets with addresses of the form 200.0.x. Give forwarding tables for each of A♢, B, C and D. Each table should include each of the listed subnets and also a **default** entry that routes traffic toward router E. Directly connected subnets may be listed with a next_hop of "direct".





5.0. (This exercise is an attempt at modeling Internet-2 routing.) Suppose sites S_1 ... S_n each have a single connection to the standard Internet, and each site S_i has a single IPv4 address block A_i. Each site's connection to the Internet is through a single router R_i; each R_i's default route points towards the standard Internet. The sites also maintain a separate, higher-speed network among themselves; each site has a single link to this separate network, also through R_i. Describe what the forwarding tables on each R_i will have to look like so that traffic from one S_i to another will always use the separate higher-speed network.

6.0. For each IPv4 network prefix given (with length), identify which of the subsequent IPv4 addresses are part of the same subnet.

(a). **10.0.130.0/23**: 10.0.130.23, 10.0.129.1, 10.0.131.12, 10.0.132.7
(b). **10.0.132.0/22**: 10.0.130.23, 10.0.135.1, 10.0.134.12, 10.0.136.7
(c). **10.0.64.0/18**: 10.0.65.13, 10.0.32.4, 10.0.127.3, 10.0.128.4
(d).♢ **10.0.168.0/21**: 10.0.166.1, 10.0.170.3, 10.0.174.5, 10.0.177.7
(e). **10.0.0.64/26**: 10.0.0.125, 10.0.0.66, 10.0.0.130, 10.0.0.62

6.5. Convert the following subnet masks to /k notation, and vice-versa:

(a).♢ 255.255.240.0
(b). 255.255.248.0
(c). 255.255.255.192
(d).♢ /20
(e). /22
(f). /27

7.0. Suppose that the subnet bits below for the following five subnets A-E all come from the beginning of the fourth byte of the IPv4 address; that is, these are subnets of a /24 block.

- A: 00
- B: 01
- C: 110
- D: 111
- E: 1010

(a). What are the sizes of each subnet, and the corresponding decimal ranges? Count the addresses with

host bits all 0's or with host bits all 1's as part of the subnet.

(b). How many IPv4 addresses in the class-C block do not belong to any of the subnets A, B, C, D and E?

8.0. In *7.9 Address Resolution Protocol: ARP* it was stated that, in newer implementations, "repeat ARP queries about a timed out entry are first sent unicast", in order to reduce broadcast traffic. Suppose multiple unicast repeat-ARP queries for host A's IP address fail, but a followup broadcast query for A's address succeeds. What probably changed at host A?

9.0. Suppose A broadcasts an ARP query "who-has B?", receives B's response, and proceeds to send B a regular IPv4 packet. If B now wishes to reply, why is it likely that A will already be present in B's ARP cache? Identify a circumstance under which this can fail.

10.0. Suppose A broadcasts an ARP request "who-has B", but inadvertently lists the physical address of another machine C instead of its own (that is, A's ARP query has IP_{src} = A, but LAN_{src} = C). What will happen? Will A receive a reply? Will any other hosts on the LAN be able to send to A? What entries will be made in the ARP caches on A, B and C?

11.0. Suppose host A connects to the Internet via Wi-Fi. The default router is R_W. Host A now begins exchanging packets with a remote host B: A sends to B, B replies, *etc*. The exact form of the connection does not matter, except that TCP may not work.

(a). You now plug in A's Ethernet cable. The Ethernet port is assumed to be on a different subnet from the Wi-Fi (so that the strong and weak end-system models of *7.9.5 ARP and multihomed hosts* do not play a role here). Assume A automatically selects the new Ethernet connection as its default route, with router R_E. What happens to the original connection to A? Can packets still travel back and forth? Does the return address used for either direction change?

(b). You now *disconnect* A's Wi-Fi interface, leaving the Ethernet interface connected. What happens now to the connection to B? Hint: to what IP address are the packets from B being sent?

See also *9 Routing-Update Algorithms*, *9 Routing-Update Algorithms* exercise 13.0, and *12 TCP Transport*, exercise 13.0.

8 IP VERSION 6

What has been learned from experience with IPv4? First and foremost, more than 32 bits are needed for addresses; the primary motive in developing IPv6 was the specter of running out of IPv4 addresses (something which, at the highest level, has already happened; see the discussion at the end of *1.10 IP - Internet Protocol*). Another important issue is that IPv4 requires (or used to require) a modest amount of effort at configuration; IPv6 was supposed to improve this.

By 1990 the IETF was actively interested in proposals to replace IPv4. A working group for the so-called "IP next generation", or IPng, was created in 1993 to select the new version; **RFC 1550** was this group's formal solicitation of proposals. In July 1994 the IPng directors voted to accept a modified version of the "Simple Internet Protocol", or SIP (unrelated to the Session Initiation Protocol, *20.11.4 RTP and VoIP*), as the basis for IPv6. The first IPv6 specifications, released in 1995, were **RFC 1883** (now **RFC 2460**, with updates) for the basic protocol, and **RFC 1884** (now **RFC 4291**, again with updates) for the addressing architecture.

SIP addresses were originally 64 bits in length, but in the month leading up to adoption as the basis for IPv6 this was increased to 128. 64 bits would probably have been enough, but the problem is less the actual number than the simplicity with which addresses can be allocated; the more bits, the easier this becomes, as sites can be given relatively large address blocks without fear of waste. A secondary consideration in the 64-to-128 leap was the potential to accommodate now-obsolete CLNP addresses (*1.15 IETF and OSI*), which were up to 160 bits in length, but compressible.

IPv6 has to some extent returned to the idea of a fixed division between network and host portions; for most IPv6 addresses, the first 64 bits is the network prefix (including any subnet portion) and the remaining 64 bits represents the host portion. The rule as spelled out in **RFC 2460**, in 1998, was that the 64/64 split would apply to all addresses except those beginning with the bits 000; those addresses were then held in reserve in the unlikely event that the 64/64 split ran into problems in the future. This was a change from 1995, when **RFC 1884** envisioned 48-bit host portions and 80-bit prefixes.

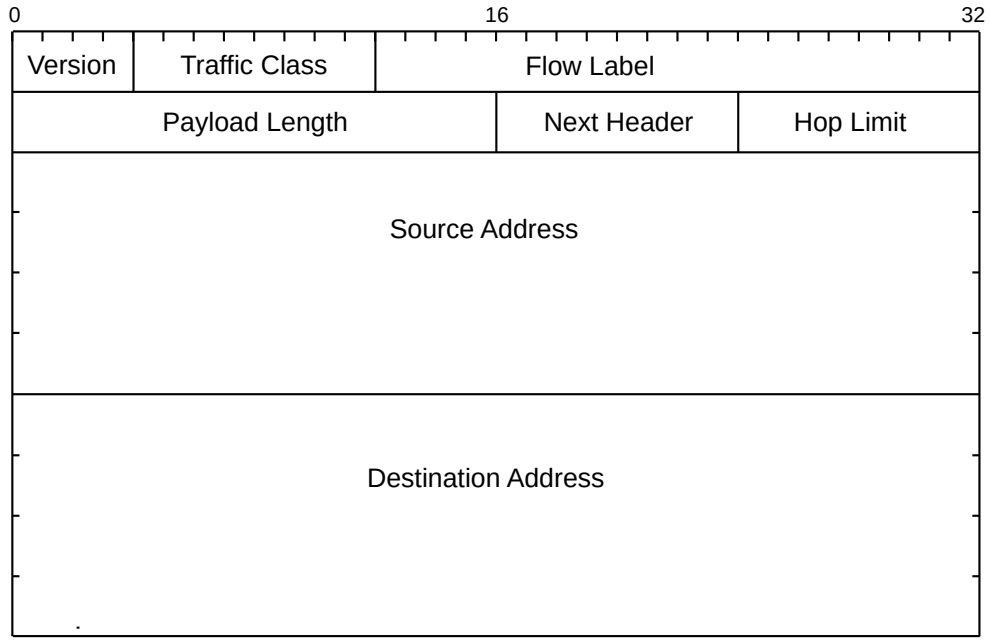
While the IETF occasionally revisits the issue, at the present time the 64/64 split seems here to stay; for discussion and justification, see *8.10.1 Subnets and /64* and **RFC 7421**. The 64/64 split is not automatic, however; there is no default prefix length as there was in the Class A/B/C IPv4 scheme. Thus, it is misleading to think of IPv6 as a return to something like IPv4's classful addressing scheme. Router advertisements must always include the prefix length, and, when assigning IPv6 addresses manually, the /64 prefix length must be specified explicitly; see *8.12.3 Manual address configuration*.

High-level routing, however, can, as in IPv4, be done on prefixes of any length (usually that means lengths shorter than /64). Routing can also be done on different prefix lengths at different points of the network.

IPv6 is now twenty years old, and yet usage as of 2015 remains quite modest. However, the shortage in IPv4 addresses has begun to loom ominously; IPv6 adoption rates may rise quickly if IPv4 addresses begin to climb in price.

8.1 The IPv6 Header

The IPv6 **fixed header** is pictured below; at 40 bytes, it is twice the size of the IPv4 header. The fixed header is intended to support only what *every* packet needs: there is no support for fragmentation, no header checksum, and no option fields. However, the concept of **extension headers** has been introduced to support some of these as options; some IPv6 extension headers are described in *8.5 IPv6 Extension Headers*. Whatever header comes next is identified by the Next Header field, much like the IPv4 Protocol field. Some other fixed-header fields have also been renamed from their IPv4 analogues: the IPv4 TTL is now the IPv6 Hop_Limit (still decremented by each router with the packet discarded when it reaches 0), and the IPv4 DS field has become the IPv6 Traffic Class.



The Flow Label is new. **RFC 2460** states that it

> may be used by a source to label sequences of packets for which it requests special handling by the IPv6 routers, such as non-default quality of service or "real-time" service.

Senders not actually taking advantage of any quality-of-service options are supposed to set the Flow Label to zero.

When used, the Flow Label represents a sender-computed hash of the source and destination addresses, and perhaps the traffic class. Routers can use this field as a way to look up quickly any priority or reservation state for the packet. All packets belonging to the same flow should have the same Routing Extension header, *8.5.3 Routing Header*. The Flow Label will in general *not* include any information about the source and destination *port* numbers, except that only some of the connections between a pair of hosts may make use of this field.

A **flow**, as the term is used here, is *one-way*; the return traffic belongs to a different flow. Historically, the term "flow" has also been used at various other scales: a single bidirectional TCP connection, multiple *related* TCP connections, or even all traffic from a particular subnet (*eg* the "computer-lab flow").

8.2 IPv6 Addresses

IPv6 addresses are written in eight groups of four hex digits, with a-f preferred over A-F (**RFC 5952**). The groups are separated by colons, and have leading 0's removed, *eg*

fedc:13:1654:310:fedc:bc37:61:3210

If an address contains a long run of 0's – for example, if the IPv6 address had an embedded IPv4 address – then when writing the address the string "::" should be used to represent however many blocks of 0000 as are needed to create an address of the correct length; to avoid ambiguity this can be used only once. Also, embedded IPv4 addresses may continue to use the "." separator:

::ffff:147.126.65.141

The above is an example of one standard IPv6 format for representing IPv4 addresses (see *8.11 Using IPv6 and IPv4 Together*). 48 bits are explicitly displayed; the :: means these are prefixed by 80 0-bits.

The IPv6 loopback address is ::1 (that is, 127 0-bits followed by a 1-bit).

Network address **prefixes** may be written with the "/" notation, as in IPv4:

12ab:0:0:cd30::/60

RFC 3513 suggested that initial IPv6 unicast-address allocation be initially limited to addresses beginning with the bits 001, that is, the 2000::/3 block (20 in binary is 0010 0000).

Generally speaking, IPv6 addresses consist of a 64-bit network prefix (perhaps including subnet bits) followed by a 64-bit "interface identifier". See *8.3 Network Prefixes* and *8.2.1 Interface identifiers*.

IPv6 addresses all have an associated **scope**, defined in **RFC 4007**. The scope of a unicast address is either **global**, meaning it is intended to be globally routable, or **link-local**, meaning that it will only work with directly connected neighbors (*8.2.2 Link-local addresses*). The loopback address is considered to have link-local scope. A few more scope levels are available for multicast addresses, *eg* "site-local" (**RFC 4291**). The scope of an IPv6 address is implicitly coded within the first 64 bits; addresses in the 2000::/3 block above, for example, have global scope.

Packets with local-scope addresses (*eg* link-local addresses) for either the destination or the source cannot be routed (the latter because a reply would be impossible).

Although addresses in the "unique local address" category of *8.3 Network Prefixes* officially have global scope, in a practical sense they still behave as if they had the now-officially-deprecated "site-local scope".

8.2.1 Interface identifiers

As mentioned earlier, most IPv6 addresses can be divided into a 64-bit network prefix and a 64-bit "host" portion, the latter corresponding to the "host" bits of an IPv4 address. These host-portion bits are known officially as the **interface identifier**; the change in terminology reflects the understanding that all IP addresses attach to interfaces rather than to hosts.

The original plan for the interface identifier was to derive it in most cases from the LAN address, though the interface identifier can also be set administratively. Given a 48-bit Ethernet address, the interface identifier based on it was to be formed by inserting 0xfffe between the first three bytes and the last three bytes, to get 64 bits in all. The seventh bit of the first byte (the Ethernet "universal/local" flag) was then set to 1.

The result of this process is officially known as the **Modified EUI-64 Identifier**, where EUI stands for Extended Unique Identifier; details can be found in **RFC 4291**. As an example, for a host with Ethernet address 00:a0:cc:24:b0:e4, the EUI-64 identifier would be 0**2**a0:ccff:fe24:b0e4 (the leading 00 becomes 02 when the seventh bit is turned on). At the time the EUI-64 format was proposed, it was widely expected that Ethernet MAC addresses would eventually become 64 bits in length.

EUI-64 interface identifiers turn out to introduce a major privacy concern: no matter where a (portable) host connects to the Internet – home or work or airport or Internet cafe – such an interface identifier always remains the same, and thus serves as a permanent host fingerprint. As a result, EUI-64 identifiers are now discouraged for personal workstations and mobile devices. (Some fixed-location hosts continue to use EUI-64 interface identifiers, or, alternatively, administratively assigned interface identifiers.)

RFC 7217 proposes an alternative: the interface identifier is a secure hash (*22.6 Secure Hashes*) of a "Net_Iface" parameter, the 64-bit IPv6 address prefix, and a host-specific secret key (a couple other parameters are also thrown into the mix, but they need not concern us here). The "Net_Iface" parameter can be the interface's MAC address, but can also be the interface's "name", *eg* `eth0`. Interface identifiers created this way change from connection point to connection point (because the prefix changes), do not reveal the Ethernet address, and are randomly scattered (because of the key, if nothing else) through the 2^{64}-sized interface-identifier space. The last feature makes probing for IPv6 addresses effectively impossible; see exercise 6.0.

Interface identifiers as in the previous paragraph do not change unless the prefix changes, which normally happens only if the host is moved to a new network. In *8.7.2.1 SLAAC privacy* we will see that interface identifiers are often changed at regular intervals, for privacy reasons.

Finally, interface identifiers are often centrally assigned, using DHCPv6 (*8.7.3 DHCPv6*).

Remote probing for IPv6 addresses based on EUI-64 identifiers is much easier than for those based on RFC-7217 identifiers, as the former are not very random. If an attacker can guess the hardware vendor, and thus the first three bytes of the Ethernet address (*2.1.3 Ethernet Address Internal Structure*), there are only 2^{24} possibilities, down from 2^{64}. As the last three bytes are often assigned in serial order, considerable further narrowing of the search space may be possible. While it may amount to security through obscurity, keeping internal global IPv6 addresses hidden is often of practical importance.

Additional discussion of host-scanning in IPv6 networks can be found in **RFC 7707** and draft-ietf-opsec-ipv6-host-scanning-06.

8.2.2 Link-local addresses

IPv6 defines **link-local** addresses, with so-called link-local scope, intended to be used only on a single LAN and never routed. These begin with the 64-bit link-local prefix consisting of the ten bits 1111 1110 10 followed by 54 more zero bits; that is, fe80::/64. The remaining 64 bits are the interface identifier for the link interface in question, above. The EUI-64 link-local address of the machine in the previous section with Ethernet address 00:a0:cc:24:b0:e4 is thus fe80::2a0:ccff:fe24:b0e4.

The main applications of link-local addresses are as a "bootstrap" address for global-address autoconfiguration (*8.7.2 Stateless Autoconfiguration (SLAAC)*), and as an optional permanent address for routers. IPv6 routers often communicate with neighboring routers via their link-local addresses, with the understanding that these do not change when global addresses (or subnet configurations) change (**RFC 4861** §6.2.8). If EUI-64 interface identifiers are used then the link-local address does change whenever the Ethernet hard-

ware is replaced. However, if **RFC 7217** interface identifiers are used and that mechanism's "Net_Iface" parameter represents the interface name rather than its physical address, the link-local address can be constant for the life of the host. (When RFC 7217 is used to generate link-local addresses, the "prefix" hash parameter is the link-local prefix fe80::/64.)

A consequence of identifying routers to their neighbors by their link-local addresses is that it is often possible to configure routers so they do not even have global-scope addresses; for forwarding traffic and for exchanging routing-update messages, link-local addresses are sufficient. Similarly, many ordinary hosts forward packets to their default router using the latter's link-local address. We will return to router addressing in *8.13.2 Setting up a router* and *8.13.2.1 A second router*.

For non-Ethernet-like interfaces, *eg* tunnel interfaces, there may be no natural candidate for the interface identifier, in which case a link-local address *may* be assigned manually, with the low-order 64 bits chosen to be unique for the link in question.

When sending to a link-local address, one must separately supply somewhere the link's "zone identifier", often by appending a string containing the interface name to the IPv6 address, *eg* fe80::f00d:cafe**%eth0**. See *8.12.1 ping6* and *8.12.2 TCP connections using link-local addresses* for examples of such use of link-local addresses.

IPv4 also has true link-local addresses, defined in **RFC 3927**, though they are rarely used; such addresses are in the 169.254.0.0/16 block (not to be confused with the 192.168.0.0/16 private-address block). Other than these, IPv4 addresses always implicitly identify the link subnet by virtue of the network prefix.

Once the link-local address is created, it must pass the **duplicate-address detection** test before being used; see *8.7.1 Duplicate Address Detection*.

8.2.3 Anycast addresses

IPv6 also introduced **anycast** addresses. An anycast address might be assigned to each of a set of routers (in addition to each router's own unicast addresses); a packet addressed to this anycast address would be delivered to only one member of this set. Note that this is quite different from multicast addresses; a packet addressed to the latter is delivered to *every* member of the set.

It is up to the local routing infrastructure to decide which member of the anycast group would receive the packet; normally it would be sent to the "closest" member. This allows hosts to send to any of a set of routers, rather than to their designated individual default router.

Anycast addresses are not marked as such, and a node sending to such an address need not be aware of its anycast status. Addresses are anycast simply because the routers involved have been configured to recognize them as such.

IPv4 anycast exists also, but in a more limited form (*10.6.7 BGP and Anycast*); generally routers are configured much more indirectly (*eg* through BGP).

8.3 Network Prefixes

We have been assuming that an IPv6 address, at least as seen by a host, is composed of a 64-bit network prefix and a 64-bit interface identifier. As of 2015 this remains a requirement; **RFC 4291** (IPv6 Addressing Architecture) states:

> For all unicast addresses, except those that start with the binary value 000, Interface IDs are required to be 64 bits long. . . .

This /64 requirement is occasionally revisited by the IETF, but is unlikely to change for mainstream IPv6 traffic. This firm 64/64 split is a departure from IPv4, where the host/subnet division point has depended, since the development of subnets, on local configuration.

Note that while the net/interface (net/host) division point is fixed, routers may still use CIDR (*10.1 Classless Internet Domain Routing: CIDR*) and may still base forwarding decisions on prefixes shorter than /64.

As of 2015, all allocations for globally routable IPv6 prefixes are part of the 2000::/3 block.

IPv6 also defines a variety of specialized network prefixes, including the link-local prefix and prefixes for anycast and multicast addresses. For example, as we saw earlier, the prefix ::ffff:0:0/96 identifies IPv6 addresses with embedded IPv4 addresses.

The most important class of 64-bit network prefixes, however, are those supplied by a provider or other address-numbering entity, and which represent the first half of globally routable IPv6 addresses. These are the prefixes that will be visible to the outside world.

IPv6 customers will typically be assigned a relatively large block of addresses, *eg* /48 or /56. The former allows 64–48 = 16 bits for local "subnet" specification within a 64-bit network prefix; the latter allows 8 subnet bits. These subnet bits are – as in IPv4 – supplied through router configuration; see *8.10 IPv6 Subnets*. The closest IPv6 analogue to the IPv4 subnet mask is that all network prefixes are supplied to hosts with an associated length, although that length will almost always be 64 bits.

Many sites will have only a single externally visible address block. However, some sites may be multihomed and thus have multiple independent address blocks.

Sites may also have private **unique local address** prefixes, corresponding to IPv4 private address blocks like 192.168.0.0/16 and 10.0.0.0/8. They are officially called Unique Local Unicast Addresses and are defined in **RFC 4193**; these replace an earlier **site-local** address plan (and official site-local scope) formally deprecated in **RFC 3879** (though unique-local addresses are sometimes still informally referred to as site-local).

The first 8 bits of a unique-local prefix are 1111 1101 (fd00::/8). The related prefix 1111 1100 (fc00::/8) is reserved for future use; the two together may be consolidated as fc00::/7. The last 16 bits of a 64-bit unique-local prefix represent the subnet ID, and are assigned either administratively or via autoconfiguration. The 40 bits in between, from bit 8 up to bit 48, represent the **Global ID**. A site is to set the Global ID to a pseudorandom value.

The resultant unique-local prefix is "almost certainly" globally unique (and is considered to have **global scope** in the sense of *8.2 IPv6 Addresses*), although it is not supposed to be routed off a site. Furthermore, a site would generally not admit any packets from the outside world addressed to a destination with the Global ID as prefix. One rationale for choosing unique Global IDs for each site is to accommodate potential later mergers of organizations without the need for renumbering; this has been a chronic problem for sites using private IPv4 address blocks. Another justification is to accommodate VPN connections from other sites. For example, if I use IPv4 block 10.0.0.0/8 at home, and connect using VPN to a site also using 10.0.0.0/8, it is possible that my printer will have the same IPv4 address as their application server.

8.4 IPv6 Multicast

IPv6 has moved away from LAN-layer *broad*cast, instead providing a wide range of LAN-layer *multi*cast groups. (Note that LAN-layer multicast is often straightforward; it is general IP-layer multicast (*20.5 Global IP Multicast*) that is problematic. See *2.1.2 Ethernet Multicast* for the Ethernet implementation.) This switch to multicast is intended to limit broadcast traffic in general, though many switches still propagate LAN multicast traffic everywhere, like broadcast.

An IPv6 multicast address is one beginning with the eight bits 1111 1111 (ff00::/8); numerous specific such addresses, and even classes of addresses, have been defined. For actual delivery, IPv6 multicast addresses correspond to LAN-layer (*eg* Ethernet) multicast addresses through a well-defined static correspondence; specifically, if x, y, z and w are the last four bytes of the IPv6 multicast address, in hex, then the corresponding Ethernet multicast address is 33:33:x:y:z:w (**RFC 2464**). A typical IPv6 host will need to join (that is, subscribe to) several Ethernet multicast groups.

The IPv6 multicast address with the broadest scope is **all-nodes**, with address ff02::1; the corresponding Ethernet multicast address is 33:33:00:00:00:01. This essentially corresponds to IPv4's LAN broadcast, though the use of LAN multicast here means that non-IPv6 hosts should not see packets sent to this address. Another important IPv6 multicast address is ff02::2, the **all-routers** address. This is meant to be used to reach all routers, and routers only; ordinary hosts do not subscribe.

Generally speaking, IPv6 nodes on Ethernets send LAN-layer **Multicast Listener Discovery** (MLD) messages to multicast groups they wish to start using; these messages allow multicast-aware Ethernet switches to optimize forwarding so that only those hosts that have subscribed to the multicast group in question will receive the messages. Otherwise switches are supposed to treat multicast like broadcast; worse, some switches may simply fail to forward multicast packets to destinations that have not explicitly opted to join the group.

8.5 IPv6 Extension Headers

In IPv4, the IP header contained a Protocol field to identify the next header; usually UDP or TCP. All IPv4 options were contained in the IP header itself. IPv6 has replaced this with a scheme for allowing an arbitrary chain of supplemental IPv6 headers. The IPv6 Next Header field *can* indicate that the following header is UDP or TCP, but can also indicate one of several IPv6 options. These optional, or extension, headers include:

- Hop-by-Hop options header
- Destination options header
- Routing header
- Fragment header
- Authentication header
- Mobility header
- Encapsulated Security Payload header

These extension headers must be processed in order; the recommended order for inclusion is as above. Most of them are intended for processing only at the destination host; the hop-by-hop and routing headers are exceptions.

8.5.1 Hop-by-Hop Options Header

This consists of a set of ⟨type,value⟩ pairs which are intended to be processed by each router on the path. A tag in the type field indicates what a router should do if it does not understand the option: drop the packet, or continue processing the rest of the options. The only Hop-by-Hop options provided by **RFC 2460** were for padding, so as to set the alignment of later headers.

RFC 2675 later defined a Hop-by-Hop option to support IPv6 **jumbograms**: datagrams larger than 65,535 bytes. The need for such large packets remains unclear, in light of *5.3 Packet Size*. IPv6 jumbograms are not meant to be used if the underlying LAN does not have an MTU larger than 65,535 bytes; the LAN world is not currently moving in this direction.

Because Hop-by-Hop Options headers must be processed by each router encountered, they have the potential to overburden the Internet routing system. As a result, **RFC 6564** strongly discourages new Hop-by-Hop Option headers, unless examination at every hop is essential.

8.5.2 Destination Options Header

This is very similar to the Hop-by-Hop Options header. It again consists of a set of ⟨type,value⟩ pairs, and the original **RFC 2460** specification only defined options for padding. The Destination header is intended to be processed at the destination, before turning over the packet to the transport layer.

Since **RFC 2460**, a few more Destination Options header types have been defined, though none is in common use. **RFC 2473** defined a Destination Options header to limit the nesting of tunnels, called the Tunnel Encapsulation Limit. **RFC 6275** defines a Destination Options header for use in Mobile IPv6. **RFC 6553**, on the Routing Protocol for Low-Power and Lossy Networks, or RPL, has defined a Destination (and Hop-by-Hop) Options type for carrying RPL data.

A complete list of Option Types for Hop-by-Hop Option and Destination Option headers can be found at www.iana.org/assignments/ipv6-parameters; in accordance with **RFC 2780**.

8.5.3 Routing Header

The original, or Type 0, Routing header contained a list of IPv6 addresses through which the packet should be routed. These did not have to be contiguous. If the list to be visited en route to destination D was ⟨R1,R2,...,Rn⟩, then this option header contained ⟨R2,R3,...,Rn,D⟩ with R1 as the initial destination address; R1 then would update this header to ⟨R1,R3,...,Rn,D⟩ (that is, the old destination R1 and the current next-router R2 were swapped), and would send the packet on to R2. This was to continue on until Rn addressed the packet to the final destination D. The header contained a Segments Left pointer indicating the next address to be processed, incremented at each Ri. When the packet arrived at D the Routing Header would contain the routing list ⟨R1,R3,...,Rn⟩. This is, in general principle, very much like IPv4 Loose Source routing. Note, however, that routers *between* the listed routers R1...Rn did not need to examine this header; they processed the packet based only on its current destination address.

This form of routing header was deprecated by **RFC 5095**, due to concerns about a traffic-amplification attack. An attacker could send off a packet with a routing header containing an alternating list of just two routers ⟨R1,R2,R1,R2,...,R1,R2,D⟩; this would generate substantial traffic on the R1–R2 link. **RFC 6275** and **RFC 6554** define more limited routing headers. **RFC 6275** defines a quite limited routing header to be used for IPv6 mobility (and also defines the IPv6 Mobility header). The **RFC 6554** routing header used for RPL, mentioned above, has the same basic form as the Type 0 header described above, but its use is limited to specific low-power routing domains.

8.5.4 IPv6 Fragment Header

IPv6 supports limited IPv4-style fragmentation via the Fragment Header. This header contains a 13-bit Fragment Offset field, which contains – as in IPv4 – the 13 high-order bits of the actual 16-bit offset of the fragment. This header also contains a 32-bit Identification field; all fragments of the same packet must carry the same value in this field.

IPv6 fragmentation is done *only* by the original sender; routers along the way are not allowed to fragment or re-fragment a packet. Sender fragmentation would occur if, for example, the sender had an 8 kB IPv6 packet to send via UDP, and needed to fragment it to accommodate the 1500-byte Ethernet MTU.

If a packet needs to be fragmented, the sender first identifies the **unfragmentable part**, consisting of the IPv6 fixed header and any extension headers that must accompany each fragment (these would include Hop-by-Hop and Routing headers). These unfragmentable headers are then attached to each fragment.

IPv6 also requires that every link on the Internet have an MTU of at least 1280 bytes beyond the LAN header; link-layer fragmentation and reassembly can be used to meet this MTU requirement (which is what ATM links (*3.5 Asynchronous Transfer Mode: ATM*) carrying IP traffic do).

Generally speaking, fragmentation should be avoided at the application layer when possible. UDP-based applications that attempt to transmit filesystem-sized (usually 8 kB) blocks of data remain persistent users of fragmentation.

8.5.5 General Extension-Header Issues

In the IPv4 world, many middleboxes (*7.7.2 Middleboxes*) examine not just the destination address but also the TCP port numbers; firewalls, for example, do this routinely to block all traffic except to a designated list of ports. In the IPv6 world, a middlebox may have difficulty *finding* the TCP header, as it must traverse a possibly lengthy list of extension headers. Worse, some of these extension headers may be newer than the middlebox, and thus unrecognized. Some middleboxes would simply drop packets with unrecognized extension headers, making the introduction of new such headers problematic.

RFC 6564 addresses this by requiring that all future extension headers use a common "type-length-value" format: the first byte indicates the extension-header's type and the second byte indicates its length. This facilitiates rapid traversal of the extension-header chain. A few older extension headers – for example the Encapsulating Security Payload header of **RFC 4303** – do not follow this rule; middleboxes must treat these as special cases.

RFC 2460 states

> With one exception [that is, Hop-by-Hop headers], extension headers are not examined or processed by any node along a packet's delivery path, until the packet reaches the node (or each of

the set of nodes, in the case of multicast) identified in the Destination Address field of the IPv6 header.

Nonetheless, sometimes intermediate nodes do attempt to add extension headers. This can break Path MTU Discovery (*12.13 Path MTU Discovery*), as the sender no longer controls the total packet size.

RFC 7045 attempts to promulgate some general rules for the real-world handling of extension headers. For example, it states that, while routers are allowed to drop packets with certain extension headers, they may not do this simply because those headers are unrecognized. Also, routers *may* ignore Hop-by-Hop Option headers, or else process packets with such headers via a slower queue.

8.6 Neighbor Discovery

IPv6 Neighbor Discovery, or **ND**, is a set of related protocols that replaces several IPv4 tools, most notably ARP, ICMP redirects and most non-address-assignment parts of DHCP. The messages exchanged in ND are part of the ICMPv6 framework, *8.9 ICMPv6*. The original specification for ND is in **RFC 2461**, later updated by **RFC 4861**. ND provides the following services:

- Finding the local router(s) [*8.6.1 Router Discovery*]
- Finding the set of network address prefixes that can be reached via local delivery (IPv6 allows there to be more than one) [*8.6.2 Prefix Discovery*]
- Finding a local host's LAN address, given its IPv6 address [*8.6.3 Neighbor Solicitation*]
- Detecting duplicate IPv6 addresses [*8.7.1 Duplicate Address Detection*]
- Determining that some neighbors are now unreachable

8.6.1 Router Discovery

IPv6 routers periodically send **Router Advertisement** (RA) packets to the all-nodes multicast group. Ordinary hosts wanting to know what router to use can wait for one of these periodic multicasts, or can request an RA packet immediately by sending a **Router Solicitation** request to the all-routers multicast group. Router Advertisement packets serve to identify the routers; this process is sometimes called **Router Discovery**. In IPv4, by comparison, the address of the default router is usually piggybacked onto the DHCP response message (*7.10 Dynamic Host Configuration Protocol (DHCP)*).

These RA packets, in addition to identifying the routers, also contain a list of all network address prefixes in use on the LAN. This is "prefix discovery", described in the following section. To a first approximation on a simple network, prefix discovery supplies the network portion of the IPv6 address; on IPv4 networks, DHCP usually supplies the entire IPv4 address.

RA packets may contain other important information about the LAN as well, such as an agreed-on MTU.

These IPv6 router messages represent a change from IPv4, in which routers need not send anything besides forwarded packets. To become an IPv4 router, a node need only have IPv4 forwarding enabled in its kernel; it is then up to DHCP (or the equivalent) to inform neighboring nodes of the router. IPv6 puts the responsibility for this notification on the router itself: for a node to become an IPv6 router, in addition to forwarding packets, it "MUST" (**RFC 4294**) also run software to support Router Advertisement. Despite this mandate, however, the RA mechanism does not play a role in the forwarding process itself; an IPv6 network can

run without Router Advertisements if every node is, for example, manually configured to know where the routers are and to know which neighbors are on-link. (We emphasize that manual configuration like this scales very poorly.)

On Linux systems, the Router Advertisement agent is most often the radvd daemon. See *8.13 IPv6 Connectivity via Tunneling* below.

8.6.2 Prefix Discovery

Closely related to Router Discovery is the **Prefix Discovery** process by which hosts learn what IPv6 network-address prefixes, above, are valid on the network. It is also where hosts learn which prefixes are considered to be local to the host's LAN, and thus reachable at the LAN layer instead of requiring router assistance for delivery. IPv6, in other words, does *not* limit determination of whether delivery is local to the IPv4 mechanism of having a node check a destination address against each of the network-address prefixes assigned to the node's interfaces.

Even IPv4 allows two IPv4 network prefixes to share the same LAN (*eg* a private one 10.1.2.0/24 and a public one 147.126.65.0/24), but a consequence of IPv4 routing is that two such LAN-sharing subnets can only reach one another via a router on the LAN, even though they should in principle be able to communicate directly. IPv6 drops this restriction.

The Router Advertisement packets sent by the router should contain a complete list of valid network-address prefixes, as the **Prefix Information** option. In simple cases this list may contain a single globally routable 64-bit prefix corresponding to the LAN subnet. If a particular LAN is part of multiple (overlapping) physical subnets, the prefix list will contain an entry for each subnet; these 64-bit prefixes will themselves likely share a common site-wide prefix of length N<64. For multihomed sites the prefix list may contain multiple unrelated prefixes corresponding to the different address blocks. Finally, site-specific "unique local" IPv6 address prefixes may also be included.

Each prefix will have an associated **lifetime**; nodes receiving a prefix from an RA packet are to use it only for the duration of this lifetime. On expiration (and likely much sooner) a node must obtain a newer RA packet with a newer prefix list. The rationale for inclusion of the prefix lifetime is ultimately to allow sites to easily **renumber**; that is, to change providers and switch to a new network-address prefix provided by a new router. Each prefix is also tagged with a bit indicating whether it can be used for autoconfiguration, as in *8.7.2 Stateless Autoconfiguration (SLAAC)* below.

Each prefix also comes with a flag indicating whether the prefix is **on-link**. If set, then every node receiving that prefix is supposed to be on the same LAN. Nodes assume that to reach a neighbor sharing the same on-link address prefix, Neighbor Solicitation is to be used to find the neighbor's LAN address. If a neighbor shares an off-link prefix, a router must be used. The IPv4 equivalent of two nodes sharing the same on-link prefix is sharing the same subnet prefix. For an example of subnets with prefix-discovery information, see *8.10 IPv6 Subnets*.

Routers advertise off-link prefixes only in special cases; this would mean that a node is part of a subnet but cannot reach other members of the subnet directly. This may apply in some wireless settings, *eg* MANETs (*3.7.8 MANETs*) where some nodes on the same subnet are out of range of one another. It may also apply when using IPv6 Mobility (*7.13 Mobile IP*, **RFC 3775**).

8.6.3 Neighbor Solicitation

Neighbor Solicitation messages are the IPv6 analogues of IPv4 ARP requests. These are essentially queries of the form "who has IPv6 address X?" While ARP requests were broadcast, IPv6 Neighbor Solicitation messages are sent to the **solicited-node multicast address**, which at the LAN layer usually represents a rather small multicast group. This address is ff02::0001:x.y.z.w, where x, y, z and w are the low-order 32 bits of the IPv6 address the sender is trying to look up. Each IPv6 host on the LAN will need to subscribe to all the solicited-node multicast addresses corresponding to its own IPv6 addresses (normally this is not too many).

Neighbor Solicitation messages are repeated regularly, but followup verifications are initially sent to the unicast LAN address on file (this is common practice with ARP implementations, but is optional). Unlike with ARP, other hosts on the LAN are not expected to eavesdrop on the initial Neighbor Solicitation message. The target host's response to a Neighbor Solicitation message is called **Neighbor Advertisement**; a host may also send these unsolicited if it believes its LAN address may have changed.

The analogue of Proxy ARP is still permitted, in that a node may send Neighbor Advertisements on behalf of another. The most likely reason for this is that the node receiving proxy services is a "mobile" host temporarily remote from the home LAN. Neighbor Advertisements sent as proxies have a flag to indicate that, if the real target does speak up, the proxy advertisement should be ignored.

Once a node (host or router) has discovered a neighbor's LAN address through Neighbor Solicitation, it continues to monitor the neighbor's continued reachability.

Neighbor Solicitation also includes Neighbor Unreachability Detection. Each node (host or router) continues to monitor its known neighbors; reachability can be inferred either from ongoing IPv6 traffic exchanges or from Neighbor Advertisement responses. If a node detects that a neighboring host has become unreachable, the original node may retry the multicast Neighbor Solicitation process, in case the neighbor's LAN address has simply changed. If a node detects that a neighboring *router* has become unreachable, it attempts to find an alternative path.

Finally, IPv4 ICMP Redirect messages have also been moved in IPv6 to the Neighbor Discovery protocol. These allow a router to tell a host that another router is better positioned to handle traffic to a given destination.

8.6.4 Security and Neighbor Discovery

In the protocols outlined above, received ND messages are trusted; this can lead to problems with nodes pretending to be things they are not. Here are two examples:

- A host can pretend to be a router simply by sending out Router Advertisements; such a host can thus capture traffic from its neighbors, and even send it on – perhaps selectively – to the real router.
- A host can pretend to be another host, in the IPv6 analog of ARP spoofing (*7.9.2 ARP Security*). If host A sends out a Neighbor Solicitation for host B, nothing prevents host C from sending out a Neighbor Advertisement claiming to be B (after previously joining the appropriate multicast group).

These two attacks can have the goal either of eavesdropping or of denial of service; there are also purely denial-of-service attacks. For example, host C can answer host B's DAD queries (below at *8.7.1 Duplicate Address Detection*) by claiming that the IPv6 address in question is indeed in use, preventing B from ever acquiring an IPv6 address. A good summary of these and other attacks can be found in **RFC 3756**.

These attacks, it is worth noting, can only be launched by nodes on the same LAN; they cannot be launched remotely. While this reduces the risk, though, it does not eliminate it. Sites that allow anyone to connect, such as Internet cafés, run the highest risk, but even in a setting in which all workstations are "locked down", a node compromised by a virus may be able to disrupt the network.

RFC 4861 suggested that, at sites concerned about these kinds of attacks, hosts might use the IPv6 Authentication Header or the Encapsulated Security Payload Header to supply digital signatures for ND packets (see *22.11 IPsec*). If a node is configured to require such checks, then most ND-based attacks can be prevented. Unfortunately, **RFC 4861** offered no suggestions beyond static configuration, which scales poorly and also rather completely undermines the goal of autoconfiguration.

A more flexible alternative is Secure Neighbor Discovery, or **SEND**, specified in **RFC 3971**. This uses public-key encryption (*22.9 Public-Key Encryption*) to validate ND messages; for the remainder of this section, some familiarity with the material at *22.9 Public-Key Encryption* may be necessary. Each message is digitally signed by the sender, using the sender's private key; the recipient can validate the message using the sender's corresponding public key. In principle this makes it impossible for one message sender to pretend to be another sender.

In practice, the problem is that public keys by themselves guarantee (if not compromised) only that the sender of a message is the same entity that previously sent messages using that key. In the second bulleted example above, in which C sends an ND message falsely claiming to be B, straightforward applications of public keys would prevent this *if* the original host A had previously heard from B, and trusted that sender to be the real B. But in general A would not know which of B or C was the real B. A cannot trust whichever host it heard from first, as it is indeed possible that C started its deception with A's very first query for B, beating B to the punch.

A common solution to this identity-guarantee problem is to create some form of "public-key infrastructure" such as **certificate authorities**, as in *22.10.2.1 Certificate Authorities*. In this setting, every node is configured to trust messages signed by the certificate authority; that authority is then configured to vouch for the identities of other nodes whenever this is necessary for secure operation. SEND implements its own version of certificate authorities; these are known as **trust anchors**. These would be configured to guarantee the identities of all routers, and perhaps hosts. The details are somewhat simpler than the mechanism outlined in *22.10.2.1 Certificate Authorities*, as the anchors and routers are under common authority. When trust anchors are used, each host needs to be configured with a list of their addresses.

SEND also supports a simpler public-key validation mechanism known as **cryptographically generated addresses**, or CGAs (**RFC 3972**). These are IPv6 interface identifiers that are secure hashes (*22.6 Secure Hashes*) of the host's public key (and a few other non-secret parameters). CGAs are an alternative to the interface-identifier mechanisms discussed in *8.2.1 Interface identifiers*. DNS names in the .onion domain used by TOR also use CGAs.

The use of CGAs makes it impossible for host C to successfully claim to be host B: only B will have the public key that hashes to B's address *and* the matching private key. If C attempts to send to A a neighbor advertisement claiming to be B, then C can sign the message with its own private key, but the hash of the corresponding public key will not match the interface-identifier portion of B's address. Similarly, in the DAD scenario, if C attempts to tell B that B's newly selected CGA address is already in use, then again C won't have a key matching that address, and B will ignore the report.

In general, CGI addresses allow recipients of a message to verify that the source address is the "owner" of the associated public key, without any need for a public-key infrastructure (*22.9.3 Trust and the Man in the Middle*). C *can* still pretend to be a router, using its own CGA address, because router addresses are not

known by the requester beforehand. However, it is easier to protect routers using trust anchors as there are fewer of them.

SEND relies on the fact that finding two inputs hashing to the same 64-bit CGA is infeasible, as in general this would take about 2^{64} tries. An IPv4 analog would be impossible as the address host portion won't have enough bits to prevent finding hash collisions via brute force. For example, if the host portion of the address has ten bits, it would take C about 2^{10} tries (by tweaking the supplemental hash parameters) until it found a match for B's CGA.

SEND has seen very little use in the IPv6 world, partly because IPv6 itself has seen such slow adoption, but also because of the perception that the vulnerabilities SEND protects against are difficult to exploit.

RA-guard is a simpler mechanism to achieve ND security, but one that requires considerable support from the LAN layer. Outlined in **RFC 6105**, it requires that each host connects directly to a switch; that is, there must be no shared-media Ethernet. The switches must also be fairly smart; it must be possible to configure them to know which ports connect to routers rather than hosts, and, in addition, it must be possible to configure them to block Router Advertisements from host ports that are *not* router ports. This is quite effective at preventing a host from pretending to be a router, and, while it assumes that the switches can do a significant amount of packet inspection, that is in fact a fairly common Ethernet switch feature. If Wi-Fi is involved, it does require that access points (which are a kind of switch) be able to block Router Advertisements; this isn't quite as commonly available. In determining which switch ports are connected to routers, **RFC 6105** suggests that there might be a brief initial learning period, during which all switch ports connecting to a device that *claims* to be a router are considered, permanently, to be router ports.

8.7 IPv6 Host Address Assignment

IPv6 provides two competing ways for hosts to obtain their full IP addresses. One is **DHCPv6**, based on IPv4's DHCP (*7.10 Dynamic Host Configuration Protocol (DHCP)*), in which the entire address is handed out by a DHCPv6 server. The other is **StateLess Address AutoConfiguration**, or SLAAC, in which the interface-identifier part of the address is generated locally, and the network prefix is obtained via prefix discovery. The original idea behind SLAAC was to support complete plug-and-play network setup: hosts on an isolated LAN could talk to one another out of the box, and if a router was introduced connecting the LAN to the Internet, then hosts would be able to determine unique, routable addresses from information available from the router.

In the early days of IPv6 development, in fact, DHCPv6 may have been intended only for address assignments to routers and servers, with SLAAC meant for "ordinary" hosts. In that era, it was still common for IPv4 addresses to be assigned "statically", via per-host configuration files. **RFC 4862** states that SLAAC is to be used when "a site is not particularly concerned with the exact addresses hosts use, so long as they are unique and properly routable."

SLAAC and DHCPv6 evolved to some degree in parallel. While SLAAC solves the autoconfiguration problem quite neatly, at this point DHCPv6 solves it just as effectively, and provides for greater administrative control. For this reason, SLAAC may end up less widely deployed. On the other hand, SLAAC gives hosts greater control over their IPv6 addresses, and so may end up offering hosts a greater degree of privacy by allowing endpoint management of the use of private and temporary addresses (below).

When a host first begins the Neighbor Discovery process, it receives a Router Advertisement packet. In this packet are two special bits: the M (managed) bit and the O (other configuration) bit. The M bit is set to

indicate that DHCPv6 is available on the network for address assignment. The O bit is set to indicate that DHCPv6 is able to provide additional configuration information (*eg* the name of the DNS server) to hosts that are using SLAAC to obtain their addresses. In addition, each individual prefix in the RA packet has an A bit, which when set indicates that the associated prefix may be used with SLAAC.

8.7.1 Duplicate Address Detection

Whenever an IPv6 host obtains a unicast address – a link-local address, an address created via SLAAC, an address received via DHCPv6 or a manually configured address – it goes through a **duplicate-address detection** (DAD) process. The host sends one or more Neighbor Solicitation messages (that is, like an ARP query), as in *8.6 Neighbor Discovery*, asking if any other host has this address. If anyone answers, then the address is a duplicate. As with IPv4 ACD (*7.9.1 ARP Finer Points*), but *not* as with the original IPv4 self-ARP, the source-IP-address field of this NS message is set to a special "unspecified" value; this allows other hosts to recognize it as a DAD query.

Because this NS process may take some time, and because addresses are in fact almost always unique, **RFC 4429** defines an **optimistic DAD** mechanism. This allows limited use of an address before the DAD process completes; in the meantime, the address is marked as "optimistic".

Outside the optimistic-DAD interval, a host is not allowed to use an IPv6 address if the DAD process has failed. **RFC 4862** in fact goes further: if a host with an established address receives a DAD query for that address, indicating that some other host wants to use that address, then the original host should discontinue use of the address.

If the DAD process fails for an address based on an EUI-64 identifier, then some other node has the same Ethernet address and you have bigger problems than just finding a working IPv6 address. If the DAD process fails for an address constructed with the **RFC 7217** mechanism, *8.2.1 Interface identifiers*, the host is able to generate a new interface identifier and try again. A counter for the number of DAD attempts is included in the hash that calculates the interface identifier; incrementing this counter results in an entirely new identifier.

While DAD works quite well on Ethernet-like networks with true LAN-layer multicast, it may be inefficient on, say, MANETs (*3.7.8 MANETs*), as distant hosts may receive the DAD Neighbor Solicitation message only after some delay, or even not at all. Work continues on the development of improvements to DAD for such networks.

8.7.2 Stateless Autoconfiguration (SLAAC)

To obtain an address via SLAAC, defined in **RFC 4862**, the first step for a host is to generate its link-local address (above, *8.2.2 Link-local addresses*), appending the standard 64-bit link-local prefix fe80::/64 to its interface identifier (*8.2.1 Interface identifiers*). The latter is likely derived from the host's LAN address using either EUI-64 or the **RFC 7217** mechanism; the important point is that it is available without network involvement.

The host must then ensure that its newly configured link-local address is in fact unique; it uses DAD (above) to verify this. Assuming no duplicate is found, then at this point the host can talk to any other hosts on the same LAN, *eg* to figure out where the printers are.

The next step is to see if there is a router available. The host may send a Router Solicitation (RS) message to the all-routers multicast address. A router – if present – should answer with a Router Advertisement

(RA) message that also contains a Prefix Information option; that is, a list of IPv6 network-address prefixes (*8.6.2 Prefix Discovery*).

As mentioned earlier, the RA message will mark with a flag those prefixes eligible for use with SLAAC; if no prefixes are so marked, then SLAAC should not be used. All prefixes will also be marked with a lifetime, indicating how long the host may continue to use the prefix. Once the prefix expires, the host must obtain a new one via a new RA message.

The host chooses an appropriate prefix, stores the prefix-lifetime information, and appends the prefix to the front of its interface identifier to create what should now be a routable address. The address so formed must now be verified through the DAD mechanism above.

In the era of EUI-64 interface identifiers, it would in principle have been possible for the receiver of a packet to extract the sender's LAN address from the interface-identifier portion of the sender's SLAAC-generated IPv6 address. This in turn would allow bypassing the Neighbor Solicitation process to look up the sender's LAN address. This was never actually permitted, however, even before the privacy options below, as there is no way to be certain that a received address was in fact generated via SLAAC. With **RFC 7217**-based interface identifiers, LAN-address extraction is no longer even potentially an option.

A host using SLAAC may receive multiple network prefixes, and thus generate for itself multiple addresses. **RFC 6724** defines a process for a host to determine, when it wishes to connect to destination address D, which of its own multiple addresses to use. For example, if D is a unique-local address, not globally visible, then the host will likely want to choose a source address that is also unique-local. **RFC 6724** also includes mechanisms to allow a host with a permanent public address (possibly corresponding to a DNS entry, but just as possibly formed directly from an interface identifier) to prefer alternative "temporary" or "privacy" addresses for outbound connections. Finally, **RFC 6724** also defines the sorting order for multiple addresses representing the same destination; see *8.11 Using IPv6 and IPv4 Together*.

At the end of the SLAAC process, the host knows its IPv6 address (or set of addresses) and its default router. In IPv4, these would have been learned through DHCP along with the identity of the host's DNS server; one concern with SLAAC is that it originally did not provide a way for a host to find its DNS server. One strategy is to fall back on DHCPv6 for this. However, **RFC 6106** now defines a process by which IPv6 routers can include DNS-server information in the RA packets they send to hosts as part of the SLAAC process; this completes the final step of autoconfiguration.

How to get DNS names for SLAAC-configured IPv6 hosts into the DNS servers is an entirely separate issue. One approach is simply not to give DNS names to such hosts. In the NAT-router model for IPv4 autoconfiguration, hosts on the inward side of the NAT router similarly do not have DNS names (although they are also not reachable directly, while SLAAC IPv6 hosts would be reachable). If DNS names are needed for hosts, then a site might choose DHCPv6 for address assignment instead of SLAAC. It is also possible to figure out the addresses SLAAC would use (by identifying the host-identifier bits) and then creating DNS entries for these hosts. Finally, hosts can also use **Dynamic DNS** (**RFC 2136**) to update their own DNS records.

8.7.2.1 SLAAC privacy

A portable host that always uses SLAAC as it moves from network to network and always bases its SLAAC addresses on the EUI-64 interface identifier (or on any other static interface identifier) will be easy to track: its interface identifier will never change. This is one reason why the obfuscation mechanism of **RFC 7217**

interface identifiers (*8.2.1 Interface identifiers*) includes the network *prefix* in the hash: connecting to a new network will then result in a new interface identifier.

Well before **RFC 7217**, however, **RFC 4941** introduced a set of **privacy extensions** to SLAAC: optional mechanisms for the generation of alternative interface identifiers, based as with RFC 7217 on pseudorandom generation using the original LAN-address-based interface identifier as a "seed" value.

RFC 4941 goes further, however, in that it supports **regular changes** to the interface identifier, to increase the difficulty of tracking a host over time even if it does not change its network prefix. One first selects a 128-bit secure-hash function F(), *eg* MD5 (*22.6 Secure Hashes*). New temporary interface IDs (IIDs) can then be calculated as follows

$$(\text{IID}_{\text{new}},\text{seed}_{\text{new}}) = \text{F}(\text{seed}_{\text{old}}, \text{IID}_{\text{old}})$$

where the left-hand pair represents the two 64-bit halves of the 128-bit return value of F() and the arguments to F() are concatenated together. (The seventh bit of IID_{new} must also be set to 0; *cf 8.2.1 Interface identifiers* where this bit is set to 1.) This process is privacy-safe even if the initial IID is based on EUI-64.

The probability of two hosts accidentally choosing the same interface identifier in this manner is vanishingly small; the Neighbor Solicitation mechanism with DAD must, however, still be used to verify that the address is in fact unique within the host's LAN.

The privacy addresses above are to be used only for connections initiated by the client; to the extent that the host accepts incoming connections and so needs a "fixed" IPv6 address, the address based on the original EUI-64/RFC-7217 interface identifier should still be available. As a result, the RFC 7217 mechanism is still important for privacy even if the RFC 4941 mechanism is fully operational.

RFC 4941 stated that privacy addresses were to be disabled by default, largely because of concerns about frequently changing IP addresses. These concerns have abated with experience and so privacy addresses are often now automatically enabled. Typical address lifetimes range from a few hours to 24 hours. Once an address has "expired" it generally remains available but *deprecated* for a few temporary-address cycles longer.

DHCPv6 also provides an option for temporary address assignments, again to improve privacy, but one of the potential advantages of SLAAC is that this process is entirely under the control of the end system.

Regularly (*eg* every few hours, or less) changing the host portion of an IPv6 address should make external tracking of a host more difficult, at least if tracking via web-browser cookies is also somehow prevented. However, for a residential "site" with only a handful of hosts, a considerable degree of tracking may be obtained simply by observing the common 64-bit prefix.

For a general discussion of privacy issues related to IPv6 addressing, see **RFC 7721**.

8.7.3 DHCPv6

The job of a DHCPv6 server is to tell an inquiring host its network prefix(es) and also supply a 64-bit host-identifier, very similar to an IPv4 DHCPv4 server. Hosts begin the process by sending a DHCPv6 request to the All_DHCP_Relay_Agents_and_Servers multicast IPv6 address ff02::1:2 (versus the broadcast address for IPv4). As with DHCPv4, the job of a relay agent is to tag a DHCPv6 request with the correct current subnet, and then to forward it to the actual DCHPv6 server. This allows the DHCPv6 server to be on a different subnet from the requester. Note that the use of multicast does nothing to diminish the need for relay agents. In fact, the All_DHCP_Relay_Agents_and_Servers multicast address scope is limited to

the current LAN; relay agents then forward to the actual DHCPv6 server using the *site*-scoped address All_DHCP_Servers.

Hosts using SLAAC to obtain their address can still use a special Information-Request form of DHCPv6 to obtain their DNS server and any other "static" DHCPv6 information.

Clients may ask for **temporary** addresses. These are identified as such in the "Identity Association" field of the DHCPv6 request. They are handled much like "permanent" address requests, except that the client may ask for a new temporary address only a short time later. When the client does so, a *different* temporary address will be returned; a repeated request for a permanent address, on the other hand, would usually return the same address as before.

When the DHCPv6 server returns a temporary address, it may of course keep a log of this address. The absence of such a log is one reason SLAAC may provide a greater degree of privacy. SLAAC also places control of the cryptographic mechanisms for temporary-address creation in the hands of the end user.

A DHCPv6 response contains a list (perhaps of length 1) of IPv6 addresses. Each separate address has an expiration date. The client must send a new request before the expiration of any address it is actually using.

In DHCPv4, the host portion of addresses typically comes from "address pools" representing small ranges of integers such as 64-254; these values are generally allocated consecutively. A DHCPv6 server, on the other hand, should take advantage of the enormous range (2^{64}) of possible host portions by allocating values more sparsely, through the use of pseudorandomness. This is to make it very difficult for an outsider who knows one of a site's host addresses to guess the addresses of other hosts, *cf* *8.2.1 Interface identifiers*.

The Internet Draft draft-ietf-dhc-stable-privacy-addresses proposes the following mechanism by which a DHCPv6 server may generate the interface-identifier bits for the addresses it hands out; F() is a secure-hash function and its arguments are concatenated together:

F(prefix, client_DUID, IAID, DAD_counter, secret_key)

The prefix, DAD_counter and secret_key arguments are as in *8.7.2.1 SLAAC privacy*. The client_DUID is the string by which the client identifies itself to the DHCPv6 server; it may be based on the Ethernet address though other options are possible. The IAID, or Identity Association identifier, is a client-provided name for this request; different names are used when requesting temporary versus permanent addresses.

Some older DHCPv6 servers may still allocate interface identifiers in serial order; such obsolete servers might make the SLAAC approach more attractive.

8.8 Globally Exposed Addresses

Perhaps the most striking difference between a contemporary IPv4 network and an IPv6 network is that on the former, many hosts are likely to be "hidden" behind a NAT router (*7.7 Network Address Translation*). On an IPv6 network, on the other hand, *every* host may be globally visible to the IPv6 world (though NAT may still be used to allow connectivity to legacy IPv4 servers).

Legacy IPv4 NAT routers provide a measure of each of privacy, security and nuisance. Privacy in IPv6 can be handled, as above, through private or temporary addresses.

The degree of security provided via NAT is entirely due to the fact that all connections must be initiated from the inside; no packet from the outside is allowed through the NAT firewall unless it is a response to a packet sent from the inside. This feature, however, can also be implemented via a conventional firewall

(IPv4 or IPv6), without address translation. Furthermore, given such a conventional firewall, it is then straightforward to modify it so as to support limited and regulated connections from the outside world as desired; an analogous modification of a NAT router is more difficult. (That said, a blanket ban on IPv6 connections from the outside can prove as frustrating as IPv4 NAT.)

Finally, one of the major reasons for hiding IPv4 addresses is that with IPv4 it is easy to map a /24 subnet by pinging or otherwise probing each of the 254 possible hosts; such mapping may reveal internal structure. In IPv6 such mapping is meant to be impractical as a /64 subnet has $2^{64} \simeq 18$ quintillion hosts (though see the randomness note in *8.2.1 Interface identifiers*). If the low-order 64 bits of a host's IPv6 address are chosen with sufficient randomness, finding the host by probing is virtually impossible; see exercise 6.0.

As for nuisance, NAT has always broken protocols that involve negotiation of new connections (*eg* TFTP, FTP, or SIP, used by VoIP); IPv6 should make these much easier to manage.

8.9 ICMPv6

RFC 4443 defines an updated version of the ICMP protocol for IPv6. As with the IPv4 version, messages are identified by 8-bit type and code (subtype) fields, making it reasonably easy to add new message formats. We have already seen the ICMP messages that make up Neighbor Discovery (*8.6 Neighbor Discovery*).

Unlike ICMPv4, ICMPv6 distinguishes between informational and error messages by the first bit of the type field. Unknown informational messages are simply dropped, while unknown error messages must be handed off, if possible, to the appropriate upper-layer process. For example, "[UDP] port unreachable" messages are to be delivered to the UDP sender of the undeliverable packet.

ICMPv6 includes an IPv6 version of Echo Request / Echo Reply, upon which the "ping6" command (*8.12.1 ping6*) is based; unlike with IPv4, arriving IPv6 echo-reply messages are delivered to the process that generated the corresponding echo request. The base ICMPv6 specification also includes formats for the error conditions below; this list is somewhat cleaner than the corresponding ICMPv4 list:

Destination Unreachable

In this case, one of the following numeric codes is returned:

0. **No route to destination**, returned when a router has no next_hop entry.
1. **Communication with destination administratively prohibited**, returned when a router *has* a next_hop entry, but declines to use it for policy reasons. Codes 5 and 6, below, are special cases of this situation; these more-specific codes are returned when appropriate.
2. **Beyond scope of source address**, returned when a router is, for example, asked to route a packet to a global address, but the return address is not, *eg* is unique-local. In IPv4, when a host with a private address attempts to connect to a global address, NAT is almost always involved.
3. **Address unreachable**, a catchall category for routing failure not covered by any other message. An example is if the packet was successfully routed to the last_hop router, but Neighbor Discovery failed to find a LAN address corresponding to the IPv6 address.
4. **Port unreachable**, returned when, as in ICMPv4, the destination host does not have the requested UDP port open.
5. **Source address failed ingress/egress policy**, see code 1.

6. **Reject route to destination**, see code 1.

Packet Too Big

This is like ICMPv4's "Fragmentation Required but DontFragment flag set"; IPv6 however has no router-based fragmentation.

Time Exceeded

This is used for cases where the Hop Limit was exceeded, and also where *source*-based fragmentation was used and the fragment-reassembly timer expired.

Parameter Problem

This is used when there is a malformed entry in the IPv6 header, an unrecognized Next Header type, or an unrecognized IPv6 option.

_node information:

8.9.1 Node Information Messages

ICMPv6 also includes **Node Information** (NI) Messages, defined in **RFC 4620**. One form of NI query allows a host to be asked directly for its name; this is accomplished in IPv4 via reverse-DNS lookups (*7.8.2 Other DNS Records*). Other NI queries allow a host to be asked for its other IPv6 addresses, or for its IPv4 addresses. Recipients of NI queries may be configured to refuse to answer.

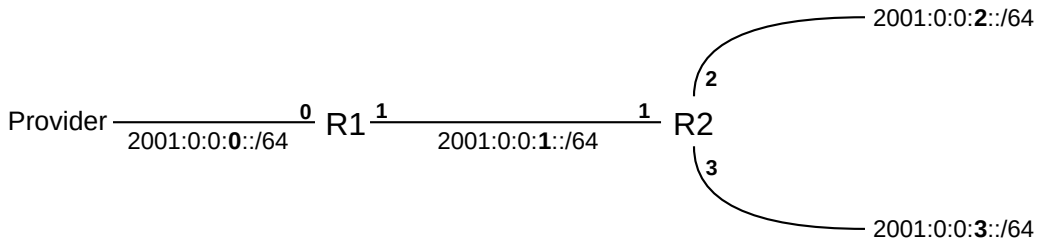
8.10 IPv6 Subnets

In the IPv4 world, network managers sometimes struggle to divide up a limited address space into a pool of appropriately sized subnets. In IPv6, this is much simpler: all subnets are of size /64, following the guidelines set out in *8.3 Network Prefixes*.

There is one common exception: **RFC 6164** permits the use of 127-bit prefixes at each end of a point-to-point link. The 128th bit is then 0 at one end and 1 at the other.

A site receiving from its provider an address prefix of size /56 can assign up to 256 /64 subnets. As with IPv4, the reasons for IPv6 subnetting are to join incompatible LANs, to press intervening routers into service as inter-subnet firewalls, or otherwise to separate traffic.

The diagram below shows a site with an external prefix of 2001::/62, two routers R1 and R2 with interfaces numbered as shown, and three internal LANS corresponding to three subnets 2001:0:0:**1**::/64, 2001:0:0:**2**::/64 and 2001:0:0:**3**::/64. The subnet 2001:0:0:**0**::/64 (2001::/64) is used to connect to the provider.



Interface 0 of R1 would be assigned an address from the /64 block 2001:0:0:**0**/64, perhaps 2001::2.

R1 will announce over its interface 1 – via router advertisements – that it has a *route* to ::/0, that is, it has the default route. It will also advertise via interface 1 the on-link prefix 2001:0:0:1::/64.

R2 will announce via interface 1 its routes to 2001:0:0:2::/64 and 2001:0:0:3::/64. It will also announce the default route on interfaces 2 and 3. On interface 2 it will advertise the on-link prefix 2001:0:0:2::/64, and on interface 3 the prefix 2001:0:0:3::/64. It could also, as a backup, advertise prefix 2001:0:0:1::/64 on its interface 1. On each subnet, only the subnet's on-link prefix is advertised.

8.10.1 Subnets and /64

Fixing the IPv6 division of prefix and host (interface) lengths at 64 bits for each is a compromise. While it does reduce the maximum number of subnets from 2^{128} to 2^{64}, in practice this is not a realistic concern, as 2^{64} is still an enormous number.

By leaving 64 bits for host identifiers, this 64/64 split leaves enough room for the privacy mechanisms of *8.7.2.1 SLAAC privacy* and *8.7.3 DHCPv6* to provide reasonable protection.

Much of the recent motivation for considering divisions other than 64/64 is grounded in concerns about ISP address-allocation policies. By declaring that users should each receive a /64 allocation, one hope is that users will in fact get enough for several subnets. Even a residential customer with only, say, two hosts and a router needs more than a single /64 address block, because the link from ISP to customer needs to be on its own subnet (it could use a 127-bit prefix, as above, but many customers would in fact have a need for multiple /64 subnets). By requiring /64 for a subnet, the hope is that users will all be allocated, for example, prefixes of at least /60 (16 subnets) or even /56 (256 subnets).

Even if that hope does not pan out, the 64/64 rule means that every user should *at least* get a /64 allocation.

On the other hand, if users *are* given only /64 blocks, and they want to use subnets, then they have to break the 64/64 rule locally. Perhaps they can create four subnets each with a prefix of length 66 bits, and each with only 62 bits for the host identifier. Wanting to do that in a standard way would dictate more flexibility in the prefix/host division.

But if the prefix/host division becomes completely arbitrary, there is nothing to stop ISPs from handing out prefixes with lengths of /80 (leaving 48 host bits) or even /120.

The general hope is that ISPs will not be so stingy with prefix lengths. But with IPv6 adoption still relatively modest, how this will all work out is not yet clear. In the IPv4 world, users use NAT (*7.7 Network Address Translation*) to create as many subnets as they desire. In the IPv6 world, NAT is generally considered to be a bad idea.

Finally, in theory it is possible to squeeze a site with two subnets onto a single /64 by converting the site's main router to a switch; all the customer's hosts now connect on an equal footing to the ISP. But this means making it much harder to use the router as a firewall, as described in *8.8 Globally Exposed Addresses*. For most users, this is too risky.

8.11 Using IPv6 and IPv4 Together

In this section we will assume that IPv6 connectivity exists at a site; if it does not, see *8.13 IPv6 Connectivity via Tunneling*.

If IPv6 coexists on a client machine with IPv4, in a so-called **dual-stack** configuration, which is used? If the client wants to connect using TCP to an IPv4-only website (or to some other network service), there is no choice. But what if the remote site also supports both IPv4 and IPv6?

The first step is the **DNS lookup**, triggered by the application's call to the appropriate address-lookup library procedure; in the Java stalk example of *11.1.3.3 The Client* we use `InetAddress.getByName()`. In the C language, address lookup is done with `getaddrinfo()` or (the now-deprecated) `gethostbyname()`. The DNS system on the client then contacts its DNS resolver and asks for the appropriate address record corresponding to the server name.

For IPv4 addresses, DNS maintains so-called "A" records, for "Address". The IPv6 equivalent is the "AAAA" record, for "Address four times longer". A dual-stack machine usually requests both. The Internet Draft draft-vavrusa-dnsop-aaaa-for-free proposes that, whenever a DNS server delivers an IPv4 A record, it also includes the corresponding AAAA record, much as IPv4 CNAME records are sent with piggybacked corresponding A records (*7.8.1 nslookup (and dig)*). The DNS requests are sent to the client's pre-configured DNS-resolver address (probably set via DHCP).

IPv6 and this book

This book is, as of April 2015, available via IPv6. Within the `cs.luc.edu` DNS zone are defined the following:

- intronetworks: both A and AAAA records
- intronetworks6: AAAA records only
- intronetworks4: A records only

DNS itself can run over either IPv4 or IPv6. A DNS server (authoritative nameserver or just resolver) using only IPv4 can answer IPv6 AAAA-record queries, and a DNS server using only IPv6 can answer IPv4 A-record queries. Ideally each nameserver would eventually support both IPv4 and IPv6 for all queries, though it is common for hosts with newly enabled IPv6 connectivity to continue to use IPv4-only resolvers. See **RFC 4472** for a discussion of some operational issues.

Here is an example of DNS requests for A and AAAA records made with the `nslookup` utility from the command line. (In this example, the DNS resolver was contacted using IPv4.)

```
nslookup -query=A facebook.com
    Name: facebook.com
```

```
        Address: 173.252.120.6
nslookup -query=AAAA facebook.com
        facebook.com has AAAA address 2a03:2880:2130:cf05:face:b00c:0:1
```

A few sites have IPv6-only DNS names. If the DNS query returns only an AAAA record, IPv6 must be used. One example in 2015 is ipv6.google.com. In general, however, IPv6-only names such as this are recommended only for diagnostics and testing. The primary DNS names for IPv4/IPv6 sites should have both types of DNS records, as in the Facebook example above (and as for google.com).

Java getByName()

The Java `getByName()` call may *not* abide by system-wide **RFC 6742**-style preferences; the Java Networking Properties documentation (2015) states that "the default behavior is to prefer using IPv4 addresses over IPv6 ones". This can be changed by setting the system property `java.net.preferIPv6Addresses` to `true`, using `System.setProperty()`.

If the client application uses a library call like Java's `InetAddress.getByName()`, which returns a *single* IP address, the client will then attempt to connect to the address returned. If an IPv4 address is returned, the connection will use IPv4, and similarly with IPv6. If an IPv6 address is returned and IPv6 connectivity is not working, then the connection will fail.

For such an application, the DNS resolver library thus effectively makes the IPv4-or-IPv6 decision. **RFC 6724**, which we encountered above in *8.7.2 Stateless Autoconfiguration (SLAAC)*, provides a configuration mechanism, through a small table of IPv6 prefixes and **precedence** values such as the following.

prefix	precedence	
::1/128	50	IPv6 loopback
::/0	40	"default" match
2002::/16	30	6to4 address; see sidebar in *8.13 IPv6 Connectivity via Tunneling*
::ffff:0:0/96	10	Matches embedded IPv4 addresses; see *8.3 Network Prefixes*
fc00::/7	3	unique-local plus reserved; see *8.3 Network Prefixes*

An address is assigned a precedence by looking it up in the table, using the longest-match rule (*10.1 Classless Internet Domain Routing: CIDR*); a list of addresses is then sorted in decreasing order of precedence. There is no entry above for link-local addresses, but by default they are ranked below global addresses. This can be changed by including the link-local prefix fe80::/64 in the above table and ranking it higher than, say, ::/0.

The default configuration is generally to prefer IPv6 if IPv6 is available; that is, if an interface has an IPv6 address that is (or should be) globally routable. Given the availability of both IPv6 and IPv4, a preference for IPv6 is implemented by assigning the prefix ::/0 – matching general IPv6 addresses – a higher precedence than that assigned to the IPv4-specific prefix ::ffff:0:0/96. This is done in the table above.

Preferring IPv6 does not always work out well, however; many hosts have IPv6 connectivity through tunneling that may be slow, limited or outright down. The precedence table can be changed to prefer IPv4 over IPv6 by raising the precedence for the prefix ::ffff:0.0.0.0/96 to a value higher than that for ::/0. Such system-wide configuration is usually done on Linux hosts by editing `/etc/gai.conf` and on Windows via the `netsh` command; for example, `netsh interface ipv6 show prefixpolicies`.

We can see this systemwide IPv4/IPv6 preference in action using OpenSSH (see *22.10.1 SSH*), between two systems that each support both IPv4 and IPv6 (the remote system here is intronetworks.cs.luc.edu). With the IPv4-matching prefix precedence set high, connection is automatically via IPv4:

/etc/gai.conf: `precedence ::ffff:0:0/96 100`
ssh: `Connecting to intronetworks.cs.luc.edu [162.216.18.28] ...`

With the IPv4-prefix precedence set low, new connections use IPv6:

/etc/gai.conf: `precedence ::ffff:0:0/96 10`
ssh: `Connecting to intronetworks.cs.luc.edu`
`[2600:3c03::f03c:91ff:fe69:f438] ...`

Applications can also use a DNS-resolver call that returns a *list* of all addresses matching a given hostname. (Often this list will have just two entries, for the IPv4 and IPv6 addresses, though round-robin DNS (*7.8 DNS*) can make the list much longer.) The C language `getaddrinfo()` call returns such a list, as does the Java `InetAddress.getAllByName()`. The **RFC 6724** preferences then determine the relative order of IPv4 and IPv6 entries in this list.

If an application requests such a list of all addresses, probably the most common strategy is to try each address in turn, according to the system-provided order. In the example of the previous paragraph, OpenSSH does in fact request a list of addresses, using `getaddrinfo()`, but, according to its source code, tries them in order and so usually connects to the first address on the list, that is, to the one preferred by the **RFC 6724** rules. Alternatively, an application might implement user-specified configuration preferences to decide between IPv4 and IPv6, though user interest in this tends to be limited (except, perhaps, by readers of this book).

The "**Happy Eyeballs**" algorithm, **RFC 8305**, offers a more nuanced strategy for deciding whether an application should connect using IPv4 or IPv6. Initially, the client might try the IPv6 address (that is, will send TCP SYN to the IPv6 address, *12.3 TCP Connection Establishment*). If that connection does not succeed within, say, 250 ms, the client would try the IPv4 address. 250 ms is barely enough time for the TCP handshake to succeed; it does not allow – and is not meant to allow – sufficient time for a retransmission. The client falls back to IPv4 well before the failure of IPv6 is certain.

IPv6 servers

As of 2015, the list of websites supporting IPv6 was modest, though the number has crept up since then. Some sites, such as apple.com and microsoft.com, require the "www" prefix for IPv6 availability. Networking providers are more likely to be IPv6-available. Sprint.com gets an honorable mention for having the shortest IPv6 address I found: 2600::aaaa.

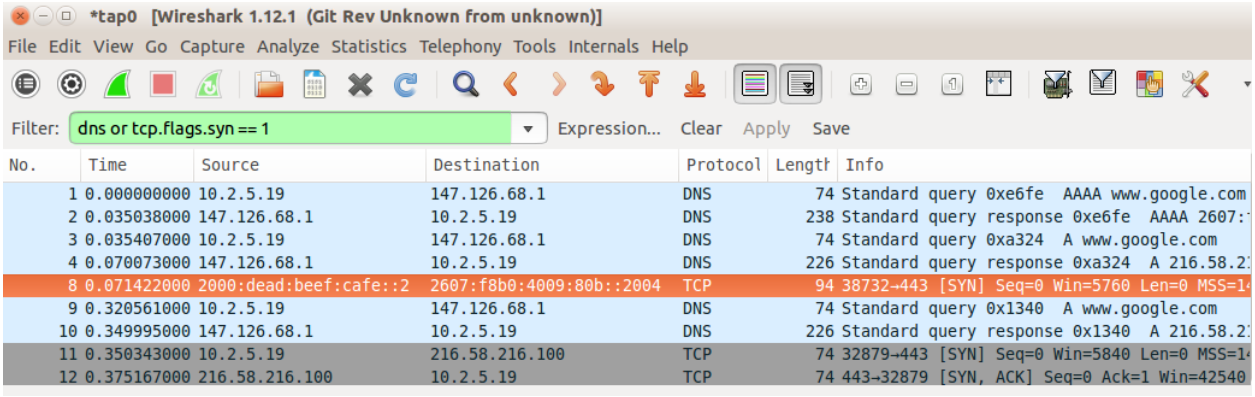
A Happy-Eyeballs client is also encouraged to **cache** the winning protocol, so for the next connection the client will attempt to use only the protocol that was successful before. The cache timeout is to be on the order of 10 minutes, so that if IPv6 connectivity failed and was restored then the client can resume using it with only moderate delay. Unfortunately, if the Happy Eyeballs mechanism is implemented at the *application* layer, which is often the case, then the scope of this cache may be limited to the particular application.

As IPv6 becomes more mainstream, Happy Eyeballs implementations are likely to evolve towards placing greater confidence in the IPv6 option. One simple change is to increase the time interval during which the client waits for an IPv6 response before giving up and trying IPv4.

We can test for the Happy Eyeballs mechanism by observing traffic with WireShark. As a first example, we imagine giving our client host a unique-local IPv6 address (in addition to its automatic link-local address); recall that unique-local addresses are not globally routable. If we now were to connect to, say, google.com, and monitor the traffic using WireShark, we would see a DNS AAAA query (IPv6) for "google.com" followed immediately by a DNS A query (IPv4). The subsequent TCP SYN, however, would be sent only to the IPv4 address: the client host would know that its IPv6 unique-local address is not routable, and it is not even tried.

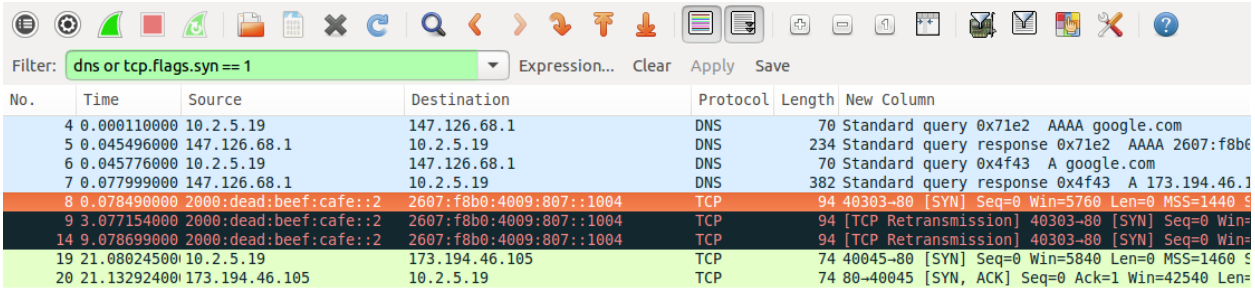
Next let us change the IPv6 address for the client host to `2000:dead:beef:cafe::2`, through manual configuration (*8.12.3 Manual address configuration*), and *without providing an actual IPv6 connection*. (We also manually specify a fake default router.) This address is part of the 2000::/3 block, and is *supposed* to be globally routable.

We now try two connections to `google.com`, TCP port 80. The first is via the Firefox browser.



We see two DNS queries, AAAA and A, in packets 1-4, followed by the first attempt (highlighted in orange) at T=0.071 to negotiate a TCP connection via IPv6 by sending a TCP SYN packet (*12.3 TCP Connection Establishment*) to the `google.com` IPv6 address 2607:f8b0:4009:80b::200e. Only 250 ms later, at T=0.321, we see a second DNS A-query (IPv4), followed by an ultimately successful connection attempt using IPv4 starting at T=0.350. This particular version of Firefox, in other words, has implemented the Happy Eyeballs dual-stack mechanism.

Now we try the connection using the previously mentioned OpenSSH application, using `-p 80` to connect to port 80. (This example was generated somewhat later; DNS now returns 2607:f8b0:4009:807::1004 as google.com's IPv6 address.)



We see two DNS queries, AAAA and A, in packets numbered 4 and 6 (pale blue); these are made by the client from its IPv4 address 10.2.5.19. Half a millisecond after the A query returns (packet 7), the client sends a TCP SYN packet to google.com's IPv6 address; this packet is highlighted in orange. This SYN packet is retransmitted 3 seconds and then 9 seconds later (in black), to no avail. After 21 seconds, the

client gives up on IPv6 and attempts to connect to `google.com` at its IPv4 address, 173.194.46.105; this connection (in green) is successful. The long delay shows that Happy Eyeballs was not implemented by OpenSSH, which its source code confirms.

(The host initiating the connections here was running Ubuntu 10.04 LTS, from 2010. The ultimately failing TCP connection gives up after three tries over only 21 seconds; newer systems make more tries and take much longer before they abandon a connection attempt.)

8.12 IPv6 Examples Without a Router

In this section we present a few IPv6 experiments that can be done without an IPv6 connection and without even an IPv6 router. Without a router, we cannot use SLAAC or DHCPv6. We will instead use link-local addresses, which require the specification of the interface along with the address, and manually configured unique-local (*8.3 Network Prefixes*) addresses. One practical problem with link-local addresses is that application documentation describing how to include a specification of the interface is sometimes sparse.

8.12.1 ping6

The IPv6 analogue of the familiar `ping` command, used to send ICMPv6 Echo Requests, is `ping6` on Linux and Mac systems and `ping -6` on Windows. The ping6 command supports an option to specify the interface; *eg* `-I eth0`; as noted above, this is mandatory when sending to link-local addresses. Here are a few ping6 examples:

ping6 ::1: This pings the host's loopback address; it should always work.

ping6 -I eth0 ff02::1: This pings the all-nodes multicast group on interface `eth0`. Here are two of the answers received:

- 64 bytes from fe80::3e97:eff:fe2c:2beb (this is the host I am pinging *from*)
- 64 bytes from fe80::2a0:ccff:fe24:b0e4 (a second Linux host)

Answers were also received from a Windows machine and an Android phone. A VoIP phone – on the same subnet but supporting IPv4 only – remained mute, despite VoIP's difficulties with IPv4 NAT that would be avoided with IPv6. In lieu of the interface option `-I eth0`, the "zone-identifier" syntax **ping6 ff02::1%eth0** also usually works; see the following section.

ping6 -I eth0 fe80::2a0:ccff:fe24:b0e4: This pings the link-local address of the second Linux host answering the previous query; again, the **%eth0** syntax should also work. The destination interface identifier here uses the now-deprecated EUI-64 format; note the "ff:fe" in the middle. Also note the flipped seventh bit of the two bytes 02a0; the destination has Ethernet address 00:a0:cc:24:b0:e4.

8.12.2 TCP connections using link-local addresses

The next experiment is to create a TCP connection. Some commands, like ping6 above, may provide for a way of specifying the interface as a command-line option. Failing that, **RFC 4007** defines the concept of a **zone identifier** that is appended to the IPv6 address, separated from it by a "%" character, to specify the link involved. On Linux systems the zone identifier is most often the interface name, *eg* `eth0` or `ppp1`.

Numeric zone identifiers are also used, in which case it represents the number of the particular interface in some designated list and can be called the **zone index**. On Windows systems the zone index for an interface can often be inferred from the output of the `ipconfig` command, which should include it with each link-local address. The use of zone identifiers is often restricted to literal (numeric) IPv6 addresses, perhaps because there is little demand for symbolic link-local addresses.

The following link-local address with zone identifier creates an ssh connection to the second Linux host in the example of the preceding section:

`ssh fe80::2a0:ccff:fe24:b0e4`**`%eth0`**

That the ssh service is listening for IPv6 connections can be verified on that host by `netstat -a | grep -i tcp6`. That the ssh connection actually *used* IPv6 can be verified by, say, use of a network sniffer like WireShark (for which the filter expression `ipv6` or `ip.version == 6` is useful). If the connection fails, but ssh works for IPv4 connections and shows as listening in the tcp6 list from the netstat command, a firewall-blocked port is a likely suspect.

8.12.3 Manual address configuration

The use of manually configured addresses is also possible, for either global or unique-local (*ie* not connected to the Internet) addresses. However, without a router there can be no Prefix Discovery, *8.6.2 Prefix Discovery*, and this may create subtle differences.

The first step is to pick a suitable prefix; in the example below we use the unique-local prefix fd37:beef:cafe::/64 (though this particular prefix does *not* meet the randomness rules for unique-local prefixes). We could also use a globally routable prefix, but here we do not want to mislead any hosts about reachability.

Without a router as a source of Router Advertisements, we need some way to specify both the prefix and the prefix *length*; the latter can be thought of as corresponding to the IPv4 subnet mask. One might be forgiven for imagining that the default prefix length would be /64, given that this is the only prefix length generally allowed (*8.3 Network Prefixes*), but this is often not the case. In the commands below, the prefix length is included at the end as the /64. This usage is just slightly peculiar, in that in the IPv4 world the slash notation is most often used only with true prefixes, with all bits zero beyond the slash length. (The Linux `ip` command also uses the slash notation in the sense here, to specify an IPv4 subnet mask, *eg* 10.2.5.37/24. The `ifconfig` and Windows `netsh` commands specify the IPv4 subnet mask the traditional way, *eg* 255.255.255.0.)

Hosts will usually assume that a prefix configured this way with a length represents an **on-link** prefix, meaning that neighbors sharing the prefix are reachable directly via the LAN.

We can now assign the low-order 64 bits manually. On Linux this is done with:

- host1: `ip -6 address add fd37:beef:cafe::1/64 dev eth0`
- host2: `ip -6 address add fd37:beef:cafe::2/64 dev eth0`

Macintosh systems can be configured similarly except the name of the interface is probably `en0` rather than `eth0`. On Windows systems, a typical IPv6-address-configuration command is

```
netsh interface ipv6 add address "Local Area Connection" fd37:beef:cafe::1/64
```

Now on host1 the command

```
ssh fd37:beef:cafe::2
```

should create an ssh connection to host2, again assuming `ssh` on host2 is listening for IPv6 connections. Because the addresses here are not link-local, /etc/host entries may be created for them to simplify entry.

Assigning IPv6 addresses manually like this is *not* recommended, except for experiments.

On a LAN not connected to the Internet and therefore with no actual routing, it is nonetheless possible to start up a Router Advertisement agent (*8.6.1 Router Discovery*), such as **radvd**, with a manually configured /64 prefix. The RA agent will include this prefix in its advertisements, and reasonably modern hosts will then construct full addresses for themselves from this prefix using SLAAC. IPv6 can then be used within the LAN. If this is done, the RA agent should also be configured to announce only a meaningless route, such as ::/128, or else nodes may falsely believe the RA agent is providing full Internet connectivity.

8.13 IPv6 Connectivity via Tunneling

The best option for IPv6 connectivity is native support by one's ISP. In such a situation one's router should be sending out Router Advertisement messages, and from these all the hosts should discover how to reach the IPv6 Internet.

If native IPv6 support is not forthcoming, however, a short-term option is to connect to the IPv6 world using **packet tunneling** (less often, some other VPN mechanism is used). **RFC 4213** outlines the common **6in4** strategy of simply attaching an IPv4 header to the front of the IPv6 packet; it is very similar to the IPv4-in-IPv4 encapsulation of *7.13.1 IP-in-IP Encapsulation*.

There are several available providers for this service; they can be found by searching for "IPv6 tunnel broker". Some tunnel brokers provide this service at no charge.

6in4, 6to4

6**in**4 tunneling should not be confused with 6**to**4 tunneling, which uses the same encapsulation as 6in4 but which constructs a site's IPv6 prefix by embedding its IPv4 address: a site with IPv4 address **129**.3.5.7 gets IPv6 prefix 2002:**81**03:0507::/48 (129 decimal = 0x81). See **RFC 3056**. There is also a 6**over**4, **RFC 2529**.

The basic idea behind 6in4 tunneling is that the tunnel broker allocates you a /64 prefix out of its own address block, and agrees to create an IPv4 tunnel to you using 6in4 encapsulation. All your IPv6 traffic from the Internet is routed by the tunnel broker to you via this tunnel; similarly, IPv6 packets from your site reach the outside world using this same tunnel. The tunnel, in other words, is your link to an IPv6 router.

Generally speaking, the MTU of the tunnel must be at least 20 bytes less than the MTU of the physical interface, to allow space for the header. At the near end this requires a local configuration change; tunnel brokers often provide a way for users to set the MTU at the far end. Practical MTU values vary from a mandatory IPv6 minimum of 1280 to the Ethernet maximum of 1500–20 = 1480.

Setting up the tunnel does not involve creating a stateful connection. All that happens is that the tunnel client (*ie* your endpoint) and the broker record each other's IPv4 addresses, and agree to accept encapsulated IPv6 packets from one another provided these two endpoint addresses are used as source and destination. The

tunnel at the client end is represented by an appropriate "virtual network interface", *eg* `sit0` or `gif0` or `IP6Tunnel`. Tunnel providers generally supply the basic commands necessary to get the tunnel interface configured and the MTU set.

Once the tunnel is created, the tunnel interface at the client end must be assigned an IPv6 address and then a (default) route. We will assume that the /64 prefix for the broker-to-client link is 2001:470:0:10::/64, with the broker at 2001:470:0:10::**1** and with the client to be assigned the address 2001:470:0:10::**2**. The address and route are set up on the client with the following commands (Linux/Mac/Windows respectively; interface names may vary, and some commands assume the interface represents a point-to-point link):

```
ip addr add 2001:470:0:10::2/64 dev sit1
ip route add ::/0 dev sit1

ifconfig gif0 inet6 2001:470:0:10::2 2001:470:0:10::1 prefixlen 128
route -n add -inet6 default 2001:470:0:10::1

netsh interface ipv6 add address IP6Tunnel 2001:470:0:10::2
netsh interface ipv6 add route ::/0 IP6Tunnel 2001:470:0:10::1
```

At this point the tunnel client should have full IPv6 connectivity! To verify this, one can use `ping6`, or visit IPv6-only versions of websites (*eg* intronetworks6.cs.luc.edu), or visit IPv6-identifying sites such as IsMyIPv6Working.com. Alternatively, one can often install a browser plugin to at least make visible whether IPv6 is used. Finally, one can use `netcat` with the `-6` option to force IPv6 use, following the HTTP example in *12.6.2 netcat again*.

There is one more potential issue. If the tunnel client is behind an IPv4 NAT router, that router must deliver arriving encapsulated 6in4 packets correctly. This can sometimes be a problem; encapsulated 6in4 packets are at some remove from the TCP and UDP traffic that the usual consumer-grade NAT router is primarily designed to handle. Careful study of the router forwarding settings may help, but sometimes the only fix is a newer router. A problem is particularly likely if two different inside clients attempt to set up tunnels simultaneously; see *7.13.1 IP-in-IP Encapsulation*.

8.13.1 IPv6 firewalls

It is strongly recommended that an IPv6 host **block new inbound connections** over its IPv6 interface (*eg* the tunnel interface), much as an IPv4 NAT router would do. Exceptions may be added as necessary for essential services (such as ICMPv6). Using the linux `ip6tables` firewall command, with IPv6-tunneled interface `sit1`, this might be done with the following:

```
ip6tables --append INPUT --in-interface sit1 --protocol icmpv6 --jump ACCEPT
ip6tables --append INPUT --in-interface sit1 --match conntrack --ctstate␣
↪ESTABLISHED,RELATED --jump ACCEPT
ip6tables --append INPUT --in-interface sit1 --jump DROP
```

At this point the firewall should be tested by attempting to access inside hosts from the outside. At a minimum, ping6 from the outside to any global IPv6 address of any inside host should fail if the ICMPv6 exception above is removed (and should succeed if the ICMPv6 exception is restored). This can be checked by using any of several websites that send pings on request; such sites can be found by searching for "online ipv6 ping". There are also a few sites that will run a remote IPv6 TCP port scan; try searching for "online ipv6 port scan". See also exercise 7.0.

8.13.2 Setting up a router

The next step, if desired, is to set up the tunnel endpoint as a router, so other hosts at the client site can also enjoy IPv6 connectivity. For this we need a second /64 prefix; we will assume this is 2001:470:0:**2**0::/64 (note this is not an "adjacent" /64; the two /64 prefixes cannot be merged into a /63). Let R be the tunnel endpoint, with `eth0` its LAN interface, and let A be another host on the LAN.

We will use the linux `radvd` package as our Router Advertisement agent (*8.6.1 Router Discovery*). In the `radvd.conf` file, we need to say that we want the LAN prefix 2001:470:0:20::/64 advertised as on-link over interface eth0:

```
interface eth0 {
        ...
        prefix 2001:470:0:20::/64
        {
                AdvOnLink on;           # advertise this prefix as on-link
                AdvAutonomous on;       # allows SLAAC with this prefix
        };
};
```

If `radvd` is now started, other LAN hosts (*eg* A) will automatically get the prefix (and thus a full SLAAC address). `Radvd` will automatically share R's default route (::/0), taking it not from the configuration file but from R's routing table. (It may still be necessary to manually configure the IPv6 address of R's eth0 interface, *eg* as 2001:470:0:20::**1**.)

On the author's version of host A, the IPv6 route is now (with some irrelevant attributes not shown)

```
default via fe80::2a0:ccff:fe24:b0e4 dev eth0
```

That is, host A routes to R via the latter's **link-local** address, always guaranteed on-link, rather than via the subnet address.

If `radvd` or its equivalent is not available, the manual approach is to assign R and A each a /64 address:

On host R: `ip -6 address add 2001:470:0:20::1/64 dev eth0`
On host A: `ip -6 address add 2001:470:0:20::2/64 dev eth0`

Because of the "/64" here (*8.12.3 Manual address configuration*), R and A understand that they can reach each other via the LAN, and do so. Host A also needs to be told of the default route via R:

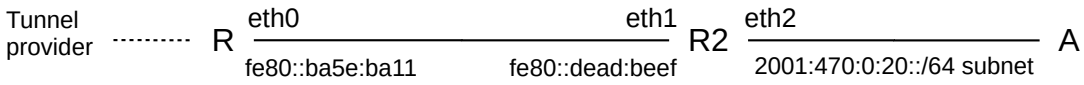
On host A: `ip -6 route add ::/0 via 2001:470:0:10::1 dev eth0`

Here we use the subnet address of R, but we could have used R's link-local address as well.

It is likely that A's `eth0` will also need its MTU configured, so that it matches that of R's virtual tunnel interface (which, recall, should be at least 20 bytes less than the MTU of R's physical outbound interface).

8.13.2.1 A second router

Now let us add a second router R2, as in the diagram below. The R—R2 link is via a separate Ethernet LAN, not a point-to-point link. The LAN with A is, as above, subnet 2001:470:0:20::/64.



In this case, it is R2 that needs to run the Router Advertisement agent (*eg* radvd). If this were an IPv4 network, the interfaces eth0 and eth1 on the R—R2 link would need IPv4 addresses from some new subnet (though the use of private addresses is an option). We can't use unnumbered interfaces (*7.12 Unnumbered Interfaces*), because the R—R2 connection is not a point-to-point link.

But with IPv6, we can configure the R—R2 routing to use only link-local addresses. Let us assume for mnemonic convenience these are as follows:

R's eth0: `fe80::ba5e:ba11`

R2's eth1: `fe80::dead:beef`

R2's forwarding table will have a default route with next_hop `fe80::ba5e:ba11` (R). Similarly, R's forwarding table will have an entry for destination subnet 2001:470:0:20::/64 with next_hop `fe80::dead:beef` (R2). Neither eth0 nor eth1 needs any other IPv6 address.

R2's eth2 interface will likely need a global IPv6 address, *eg* 2001:470:0:20::1 again. Otherwise R2 may not be able to determine that its eth2 interface is in fact connected to the 2001:470:0:20::/64 subnet.

One advantage of not giving eth0 or eth1 global addresses is that it is then impossible for an outside attacker to reach these interfaces directly. It also saves on subnets, although one hopes with IPv6 those are not in short supply. All routers at a site are likely to need, for management purposes, an IP address reachable throughout the site, but this does not have to be globally visible.

8.14 IPv6-to-IPv4 Connectivity

What happens if you switch to IPv6 completely, perhaps because your ISP (or phone provider) has run out of IPv4 addresses? Some of the time – hopefully more and more of the time – you will only need to talk to IPv6 servers. For example, the DNS names `facebook.com` and `google.com` each correspond to an IPv4 address, but also to an IPv6 address (above). But what do you do if you want to reach an IPv4-only server? Such servers are expected to continue operating for a long time to come. It is necesary to have some sort of centralized IPv6-to-IPv4 **translator**.

An early strategy was NAT-PT (**RFC 2766**). The translator was assigned a /96 prefix. The IPv6 host would append to this prefix the 32-bit IPv4 address of the destination, and use the resulting address to contact the IPv4 destination. Packets sent to this address would be delivered via IPv6 to the translator, which would translate the IPv6 header into IPv4 and then send the translated packet on to the IPv4 destination. As in IPv4 NAT (*7.7 Network Address Translation*), the reverse translation will typically involve TCP port numbers to resolve ambiguities. This approach requires the IPv6 host to be aware of the translator, and is limited to TCP and UDP (because of the use of port numbers). Due to these and several other limitations, NAT-PT was formally deprecated in **RFC 4966**.

Do you still have IPv4 service?

As of 2017, several phone providers have switched many of their customers to IPv6 while on their mobile-data networks. The change can be surprisingly inconspicuous. Connections to IPv4-only services still work just fine, courtesy of NAT64. About the only way to tell is to look up the phone's IP address.

The replacement protocol is **NAT64**, documented in **RFC 6146**. This is also based on address translation, and, as such, cannot allow connections initiated from IPv4 hosts to IPv6 hosts. It is, however, transparent to both the IPv6 and IPv4 hosts involved, and is not restricted to TCP (though only TCP, UDP and ICMP are supported by **RFC 6146**). It uses a special DNS variant, DNS64 (**RFC 6147**), as a companion protocol.

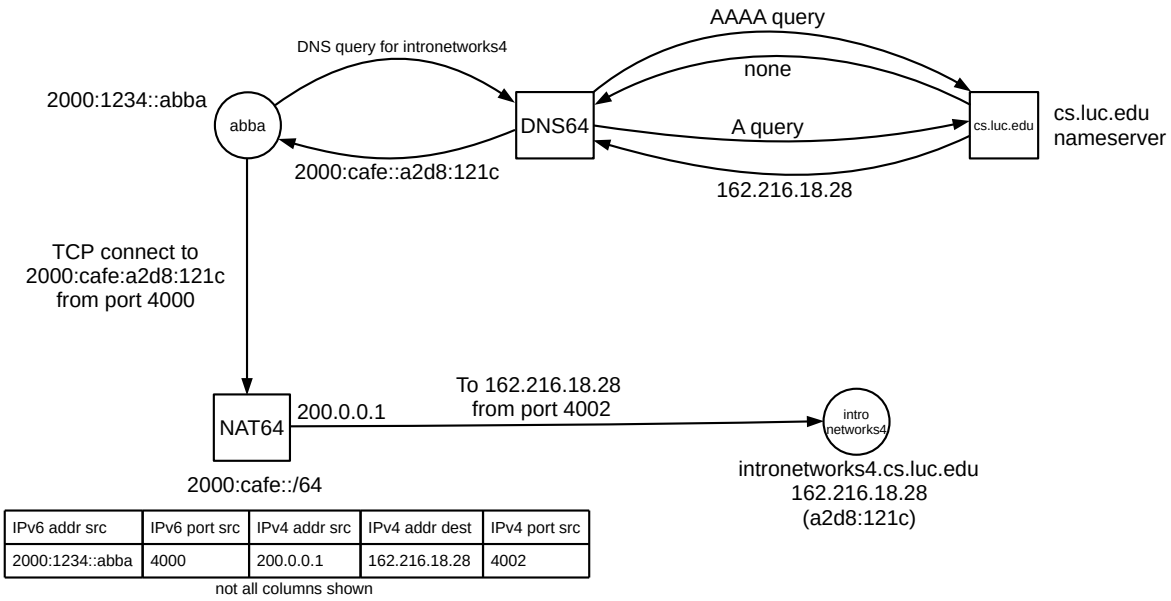
To use NAT64, an IPv6 client sends out its ordinary DNS query to find the addresses of the destination server. The DNS resolver (*7.8 DNS*) receiving the request must use DNS64. If the destination has only an IPv4 address, then the DNS resolver will return to the IPv6 client (as an AAAA record) a synthetic IPv6 address consisting of a prefix and the embedded IPv4 address of the server, much as in NAT-PT above (though multiple prefix-length options exist; see **RFC 6052**). The prefix belongs to the actual NAT64 translator; any packet addressed to an IPv6 address starting with the prefix will be delivered to the translator. There is no relationship between the NAT64 translator and the DNS64 resolver beyond the fact that the former's prefix is configured into the latter.

The IPv6 client now uses this synthetic IPv6 address to contact the IPv4 server. Its packets will be routed to the NAT64 translator itself, by virtue of the prefix, much as in NAT-PT. Upon receiving the first packet from the IPv6 client, the NAT64 translator will assign one of its IPv4 addresses to the new connection. As IPv4 addresses are in short supply, this pool of available IPv4 addresses may be small, so NAT64 allows one IPv4 address to be used by many IPv6 clients. To this end, the NAT64 translator will also (for TCP and UDP) establish a port mapping between the incoming IPv6 source port and a port number allocated by the NAT64 to ensure that traffic is uniquely reversable. As with IPv4 NAT, if two IPv6 clients try to contact the same IPv4 server using the same source ports, and are assigned the same NAT64 IPv4 address, then one of the clients will have its port number changed.

If an ICMP query is being sent, the Query Identifier is used in lieu of port numbers. To extend NAT64 to new protocols, an appropriate analog of port numbers must be identified, to allow demultiplexing of multiple connections sharing a single IPv4 address.

After the translation is set up, by creating appropriate table entries, the translated packet is sent on to the IPv4 server address that was embedded in the synthetic IPv6 address. The source address will be the assigned IPv4 address of the translator, and the source port will have been rewritten in accordance with the new port mapping. At this point packets can flow freely between the original IPv6 client and its IPv4 destination, with neither endpoint being aware of the translation (unless the IPv6 client carefully inspects the synthetic address it receives via DNS64). A timer within the NAT64 translator will delete the association between the IPv6 and IPv4 addresses if the connection is not used for a while.

As an example, suppose the IPv6 client has address 2000:1234::abba, and is trying to reach *intronetworks4.cs.luc.edu* at TCP port 80. It contacts its DNS server, which finds no AAAA record but IPv**4** address 162.216.18.28 (in hex, a2d8:121c). It takes the prefix for its NAT64 translator, which we will assume is 2000:cafe::, and returns the synthetic address 2000:cafe::a2d8:121c.



The IPv6 client now tries to connect to 2000:cafe::a2d8:121c, using source port 4000. The first packet arrives at the NAT64 translator, which assigns the connection the outbound IPv4 address of 200.0.0.1, and reassigns the source port on the IPv4 side to 4002. The new IPv4 packet is sent on to 162.216.18.28. The reply from *intronetworks4.cs.luc.edu* comes back, to ⟨200.0.0.1,4002⟩. The NAT64 translator looks this up and finds that this corresponds to ⟨2000:1234::abba,4000⟩, and forwards it back to the original IPv6 client.

8.15 Epilog

IPv4 has run out of large address blocks, as of 2011. IPv6 has reached a mature level of development. Most common operating systems provide excellent IPv6 support.

Yet conversion has been slow. Many ISPs still provide limited (to nonexistent) support, and inexpensive IPv6 firewalls to replace the ubiquitous consumer-grade NAT routers are just beginning to appear. Time will tell how all this evolves. However, while IPv6 has now been around for twenty years, top-level IPv4 address blocks disappeared much more recently. It is quite possible that this will prove to be just the catalyst IPv6 needs.

8.16 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. Each IPv6 address is associated with a specific solicited-node multicast address.

(a). Explain why, on a typical Ethernet, if the original IPv6 host address was obtained via SLAAC then the LAN multicast group corresponding to the host's solicited-node multicast addresses is likely to be small, in many cases consisting of one host only. (Packet delivery to small LAN multicast groups can be much more efficient than delivery to large multicast groups.)