

What has been learned from experience with IPv4? First and foremost, more than 32 bits are needed for addresses; the primary motive in developing the new version of IP known as 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.

In this chapter we outline the basic format of IPv6 packets, including address format and address assignment. The following chapter continues with additional features of IPv6.

By 1990 the IPv4 address-space issue was well understood, and 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 Plus”, or SIPP ([RFC 1710](#)), 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.

SIPP 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.

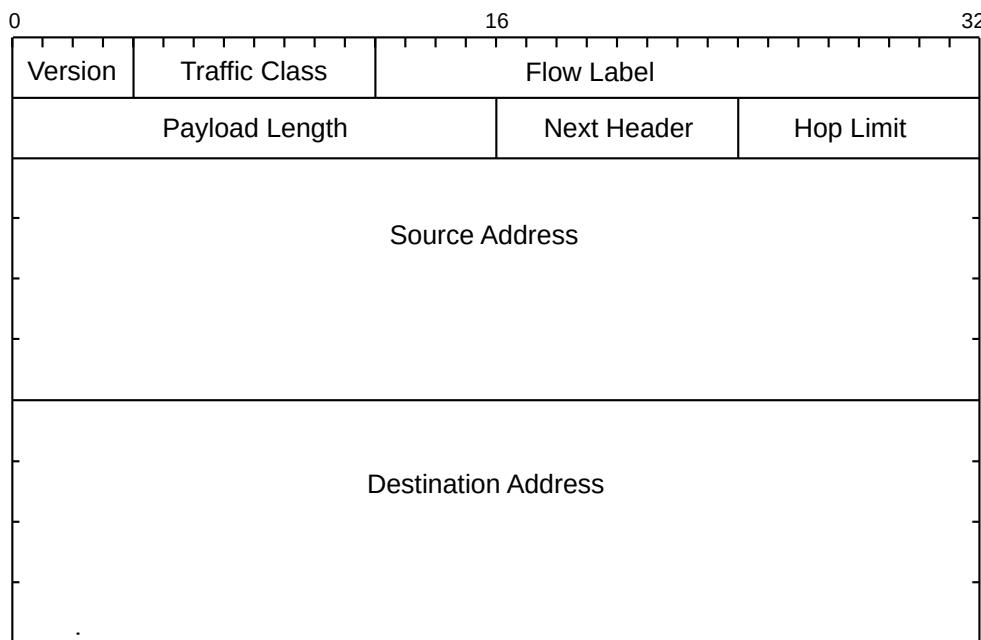
While the IETF occasionally revisits the issue, at the present time the 64/64 split seems here to stay; for discussion and justification, see [12.3.1 Subnets and /64](#) and [RFC 7421](#). The 64/64 split is not automatic, however; there is no default prefix length as there was in the Class A/B/C IPv4 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 [12.5.3 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.

11.1 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 [11.5 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, [11.5.3 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”).

11.2 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 [12.4 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 [11.3 Network Prefixes](#) and [11.2.1 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 ([11.2.2 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 [11.3 Network Prefixes](#) officially have global scope, in a practical sense they still behave as if they had the now-officially-deprecated “site-local scope”.

11.2.1 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 0xffffe 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 02a0: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 have long been recognized as 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.)

While these general issues are alone enough to warrant abandoning EUI-64 identifiers, there are, in fact, much more serious risks, such as the **IPvSeeYou** vulnerability of [RB22]. Consider a budget home router combined with Wi-Fi access point that uses EUI-64 addresses; many such devices remain on the market, and many more are currently in use and offer no upgrade path. There is an excellent chance that all MAC addresses for this router – both Ethernet and Wi-Fi – are assigned sequentially (or they are related in a way determined by the OUI). Even if a user is using a privacy-protecting interface identifier when connecting the internet, the router’s MAC address can be exposed: an IPv6 traceroute to the user’s IPv6 address will reveal the router’s public-facing IPv6 address, as the last hop before the user’s own address. From this – via EUI-64 – the public-facing Ethernet MAC address is easily found. This router MAC address, in turn, determines the router’s Wi-Fi MAC address to within a handful of values. Finally, *maps exist of the physical GPS location of almost all Wi-Fi MAC addresses*, obtained via so-called “war-driving” (driving around scanning for Wi-Fi-access-point MAC addresses and recording the GPS coordinates of each); see, for example, [wigle.net](#). Thus, from the user’s non-EUI privacy-implementing IPv6 address, the user’s real home location is straightforward to determine.

[RFC 7217](#) proposes a privacy-improving alternative to EUI-64 identifiers: the interface identifier is a secure hash ([28.6 Secure Hashes](#)) of a so-called “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 4.0.

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 [11.7.2.1 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 ([11.7.3 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](#).

11.2.2 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 ([11.7.2 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 hardware 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 [12.6.2 Setting up a router](#) and [12.6.2.1 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 [12.5.1 ping6](#) and [12.5.2 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 [11.7.1 Duplicate Address Detection](#).

11.2.3 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 ([15.8 BGP and Anycast](#)); generally routers are configured much more indirectly (*eg* through BGP).

11.3 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 ([14.1 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 [12.3 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 [11.2 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.

11.4 IPv6 Multicast

IPv6 has moved away from LAN-layer *broadcast*, instead providing a wide range of LAN-layer *multicast* groups. (Note that LAN-layer multicast is often straightforward; it is general IP-layer multicast ([25.5 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.

11.5 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.

11.5.1 Hop-by-Hop Options Header

This consists of a set of $\langle \text{type}, \text{value} \rangle$ 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 [7.3 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.

11.5.2 Destination Options Header

This is very similar to the Hop-by-Hop Options header. It again consists of a set of $\langle \text{type}, \text{value} \rangle$ 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](#).

11.5.3 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 $\langle R_1, R_2, \dots, R_n \rangle$, then this option header contained $\langle R_2, R_3, \dots, R_n, D \rangle$ with R_1 as the initial destination address; R_1 then would update this header to $\langle R_1, R_3, \dots, R_n, D \rangle$ (that is, the old destination R_1 and the current next-router R_2 were swapped), and would send the packet on to R_2 . This was to continue on until R_n 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 R_i . When the packet arrived at D the Routing Header would contain the routing list $\langle R_1, R_3, \dots, R_n \rangle$. This is, in general principle, very much like IPv4 Loose Source routing. Note, however, that routers *between* the listed routers $R_1 \dots R_n$ 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 $\langle R_1, R_2, R_1, R_2, \dots, R_1, R_2, D \rangle$; this would generate substantial traffic on the R_1 – R_2 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.

11.5.4 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 ([5.5 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.

11.5.5 General Extension-Header Issues

In the IPv4 world, many middleboxes ([9.7.2 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 facilitates 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 ([18.6 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.

11.6 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, [12.2 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) [[11.6.1 Router Discovery](#)]
- Finding the set of network address prefixes that can be reached via local delivery (IPv6 allows there to be more than one) [[11.6.2 Prefix Discovery](#)]
- Finding a local host’s LAN address, given its IPv6 address [[11.6.3 Neighbor Solicitation](#)]
- Detecting duplicate IPv6 addresses [[11.7.1 Duplicate Address Detection](#)]
- Determining that some neighbors are now unreachable

11.6.1 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 ([10.3 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 [12.6 IPv6 Connectivity via Tunneling](#) below.

11.6.2 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 [11.7.2 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 [12.3 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 ([4.2.8 MANETs](#)) where some nodes on the same subnet are out of range of one another. It may also apply when using IPv6 Mobility ([9.9 Mobile IP, RFC 3775](#)).

11.6.3 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::0001:255.y.z.w, where y, z and w are the low-order three bytes of the IPv6 address the sender is trying to look up (note that we are using the notation here for an embedded IPv4 address, even though y, z and w are from an IPv6 address). 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.

11.6.4 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 ([10.2.2 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 [11.7.1 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 [29.6 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 ([29 Public-Key Encryption](#)) to validate ND messages; for the remainder of this section, some familiarity with the material at [29 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 [29.5.2.1 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 [29.5.2.1 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 ([28.6 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 [11.2.1 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 ([29.3 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.

11.7 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 ([10.3 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 (*e.g.* 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.

11.7.1 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 [11.6 Neighbor Discovery](#), asking if any other host has this address. If anyone answers, then the address is a duplicate. As with IPv4 ACD ([10.2.1 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, [11.2.1 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 ([4.2.8 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.

11.7.2 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, [11.2.2 Link-local addresses](#)), appending the standard 64-bit link-local prefix fe80::/64 to its interface identifier ([11.2.1 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 ([11.6.2 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 at least one address per prefix. [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; see, for example, [11.7.2.1 SLAAC privacy](#). Finally, [RFC 6724](#) also defines the sorting order for multiple addresses representing the same destination; see [12.4 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.

11.7.2.1 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 ([11.2.1 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(), *e.g.* MD5 ([28.6 Secure Hashes](#)). New temporary interface IDs (IIDs) can then be calculated as follows

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

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.* [11.2.1 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.

A consequence of privacy addresses (for either SLAAC or DHCPv6) is that one host will typically have

multiple active addresses for any one network prefix, at any given time. [RFC 7934](#) suggests that a host might change its address, for privacy reasons, once per day, and that each address would have a lifetime of seven days. Add to that the use of separate addresses for virtual machines, and perhaps also for containerized applications, and [RFC 7934](#) suggests that up to 20 addresses might be needed. The number might be quite a bit higher; some proposals for privacy addresses suggest changing them much more often than once a day (though the address lifetimes might also be reduced). It would not be entirely unreasonable, in fact, for a browser to use a separate IPv6 address for each separate website accessed. The use of too many addresses does add to the memory and traffic requirements of router Neighbor Discovery ([11.6 Neighbor Discovery](#)), however.

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](#).

11.7.3 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 DHCPv6 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. Temporary addresses are typically used to improve privacy, by making it more difficult to track users by IPv6 address.

When the DHCPv6 server returns a temporary address, it may of course keep a log of this address. When SLAAC is used, a log is still possible, as each new address must run through the Neighbor Discovery ([11.6 Neighbor Discovery](#)) process. However, SLAAC does place control of the cryptographic mechanisms for temporary-address creation in the hands of the end user, rather than in a centralized service. For example, the DHCPv6 temporary-address mechanism might have a flaw that would allow a remote observer to infer a relationship between different temporary addresses, though the secure-hash mechanism described below appears to be secure as long as the `secret_key` portion is not compromised.

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 in part 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 [11.2.1 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 [11.7.2.1 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.

11.8 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.

11.9 Exercises

Exercises may be 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.)
 - (b). What steps might a DHCPv6 server take to ensure that, for the IPv6 addresses it hands out, the LAN multicast groups corresponding to the host addresses’ solicited-node multicast addresses will be small?
- 2.0. If an attacker sends a large number of probe packets via IPv4, you can block them by blocking the attacker’s IP address. Now suppose the attacker uses IPv6 to launch the probes; for each probe, the attacker