IPv6 for IPv4 Experts

Yar Tikhiy

December 31, 2013

License

Contents

List of Figures

List of Tables

Preface

Verbum sat sapienti est.
(A word to the wise is enough.)

Latin proverb

Although IPv6 has been around for quite a while now, many network engineers are still hesitant about embracing it, their valid excuse being a mere lack of time to study the new architecture. Granted, everyone knows of the legendary 128-bit IPv6 address but, unfortunately, that's about it. The irony is, now that the IPv4 address space has been exhausted and everyone seems raring to go IPv6, the rush of a forced and hurried transition does even less than anything else to encourage network engineers in real understanding of the new protocol's benefits. Indeed, who will want to spend their nights poring over the fine details if all that is needed for the moment—ASAP!—is just a longer IP address? This turn of events is hardly fair to anyone. On the one hand, the enormous effort by the creators of IPv6 to build a new Layer 3 architecture for the Internet still goes largely unrecognized. On the other hand, IT engineers, forever stuck in a hi-tech rat race, are denied the joy and excitement a true appreciation of the new technology could bring, not to mention its efficient use being just impossible without thorough understanding.

Paradoxically, among the most disadvantaged with respect to getting on the IPv6 bandwagon can be the active experts of the TCP/IP industry. First, they are just too busy to study IPv6 from scratch because everyone is after their time. Second, it would be unexciting and, for that matter, inefficient for them to try catching up with IPv6 by a basic textbook on the subject because they don't need it explained to them what a byte or a packet is.

Having shared that experience in full, we have got a more enjoyable game to offer to the old guard. In this book, we are going to follow the IPv6 architects in their footsteps and pretend that it is us—teamed up with the reader, of course—who are to design and build the entire IPv6 machine virtually from scratch, having our IPv4 experience as the only foundation. Choosing to explore IPv6 this way is not an attempt to steal the IPv6 developers' fame but a tribute to their genius, for we will soon see at first hand that almost every aspect of IPv6 can be understood and explained on a clear technical basis. Thanks to the industry's long hesitance, the IPv6 folks have had plenty of time to perfect each detail, and if IPv4 was a creation by pioneers exploring a completely new area and having no ready answers to many a question, IPv6 is a refined mechanism with each part in its proper place.

Our hope is that this unconventional approach to the subject can help to have everyone finally get what they have long deserved. First of all, the IPv4 experts seeking IPv6 knowledge will be well rewarded for their studies: They are going to enjoy new exciting ideas, to reconfirm their own TCP/IP expertise, and to gain, in almost no time, detailed knowledge and solid understanding of the IPv6 inner workings. With the demand for IPv6 on the rise, knowledge and understanding by individual engineers can pave the way for a worldwide technical culture based on appreciation and thoughtful use of the IPv6 features. Ultimately, building such a culture will make the highest tribute the IT engineering community can ever pay to the creators of IPv6.

Chapter 1

Defining the Problem

> The revolution is not a social dinner, a literary event, a drawing, or an embroidery; it cannot be done with elegance and courtesy. The revolution is an act of violence.
>
> Mao Tse-Tung

To start developing a new Internet Protocol,[1] we need a really good case for it because investments of all sorts into the current one, IP version 4, are vast. This is where IPv4 address space depletion [1] can come handy; indeed, it is hard to think of a better pretext to develop a new protocol than running out of addresses in the current one. However, our trick, as well as a challenge for us, will be to start off from the basic need for a larger IP address space but let it not govern our technical imagination.

Before using a thing we engineers had better understand it first, so let's see what IPv4 address space depletion actually means. There hardly are 2^{32} live IPv4 addresses out there in the Internet. What happened to the rest of them then? We have got to solve this missing address mystery in order to avoid the same problem in the new protocol we are to design.

It is common knowledge that a number of IPv4 addresses are reserved for special purposes [2] such as private use, documentation examples, benchmarks, and multicast. Quite a few IPv4 addresses are lost to subnetting as the subnet address and the broadcast address generally cannot be allocated—unless an unconventional */31* prefix is in use [3]. Many IPv4 addresses fell

[1] Recall that for our purposes **IP** stands for **Internet Protocol** rather than "intellectual property".

victim to overly generous allocation in the younger years of the Internet, when few people anticipated address shortage; this over-allocation was partly due to classful routing, which dominated the early Internet.

However, the greatest number of IPv4 addresses turn out to be wasted due to the hierarchical model of IP unicast address allocation [4]. As we all know, IPv4 unicast addresses were not handed out one at a time by a single authority: Rather, they were hierarchically delegated in blocks—of powers of 2, of course, which will be of relevance to us in Section 2.1. *IANA* started with the complete IPv4 address space and parceled it out to RIRs, who used their allocations to sub-delegate smaller blocks to LIRs, and so on down to a subnet [33, Section 4.2]. This scheme was very simple and easy to understand; it had a downside though: the overhead.

The problem's essence can be grasped as follows. No admin in their right mind at any tier of the pyramid would let their address stock run dry: Everyone has a few address blocks put by for a rainy day, so to speak; and the higher the tier, the larger an average block size and the greater the toll on the global address space. To be specific, because address blocks were subdivided for further allocation downstream, the amount wasted grows exponentially along with the average block size as we go back up the pyramid.

Apparently there was no way around it; this is the price we paid for not having to receive every single IPv4 address from the hands of *IANA*. Since we couldn't seem to avoid this issue, let's at least try to quantify it. Fortunately for us, this problem is not new and affects not only data networks, so we aren't the first to have to tackle it. In particular, the public switched telephone networks (**PSTN**) have suffered from it too and got some experience to share. Indeed, telephone numbers are allocated in a very similar manner, just using decimal rather than binary system.

How can we quantify address space utilization by a single index independent of the address space size? A simple used-to-total ratio won't have the feature we are after as it grows linearly with the number of addresses used. A possible trick is to use logarithms in place of absolute values for the numbers of addresses used and available in the address space. The modified ratio is known as the **Host-Density** (HD) ratio and computed as follows [4]:

$$HD = \frac{logN_{used}}{logN_{total}} \tag{1.1}$$

Quite obviously, the logarithm base choice can be arbitrary here as long as it is the same in the numerator and the denominator.

The practical experience cited in [4] proved that a traditional network such as a PSTN could no longer grow when its HD ratio got as high as 0.87. If the entire IPv4 address space were available to hosts ($N_{total} = 2^{32}$), it would take as few as 240 million hosts for the Internet to reach its critical host density level, although less than 6% addresses would actually be used.

An exercise to the reader will be to plot the graphs of the used-to-total ratio $R = \frac{N_{used}}{N_{total}}$ and the HD ratio in the same scale for $N_{total} = 2^{32}$ and to see how they differ. (Hint: *To superpose those two graphs, use the fact that both R and HD reach* 1 *when the address space is completely used up.*)

And what are the actual numbers? It is hard to tell how many IPv4 addresses are live, but it can be estimated that 2 to 3 billion addresses have been allocated to existing networks [6]. The Internet has clearly outgrown its original address space.

At the same time, it speaks very highly of the Internet and the address delegation procedures it adopted that it had been able to utilize so many IPv4 addresses before the real trouble started and the IPv4 address space was declared exhausted, reaching an unprecedented HD value of 0.96–0.98.

There even is an opinion that planning and controlling the growth of an IP network based on fixed HD thresholds taken from the historical data can be overly conservative and underestimate the potential of the IP technology to sustain densely populated networks [5].

Note well that the HD ratio will make sense only if applied to a hierarchical system. If our distribution system has just one level in it, nothing will prevent us from giving out all addresses or whatever we are to distribute. Imagine for instance that we have a crate of matches packed as follows: 100 cartons per crate, each carton with 10 matchboxes, which contain 50 matches apiece. If we start handing out the matches one by one, insisting they be lit right away, eventually there will be a pile of 50,000 burnt matches in front of us, proving a 100% utilization of the resource. But if, instead, we start handing out cartons of matchboxes, thus delegating the further distribution of matchboxes and then individual matches to the recipients, we may be sure quite a few matches will remain safe in matchboxes and cartons because their new owners will lay them away just in case.

The ill consequences of the IPv4 address shortage are many, taking their toll on both the everyday practice and the philosophy of the Internet. As replenishing IPv4 address pools is becoming increasingly problematic, network admins go to great lengths to make do with what global addresses they have already got. This gives rise to solutions enabling even a large network[2] to be represented in the Internet with just a few globally routed IPv4 addresses, the most popular devices being NAT and proxies in combination with private-use address space. Their wide deployment, in turn, can't but result in an implicit dismissal of a fundamental principle of the Internet: its end-to-end transparency [7] whereby any host A can send an IP packet to any host B, where B can be equal to A as well as different. In TCP/IP, this principle holds only as long as each host has a unique global address.

> Here is a slightly different way to put the same idea: For end-to-end transparency to materialize in the Internet, each host needed a unique global address. Indeed, as soon as it was the case, the address of host B was the only information host A ever needed to unambiguously send a packet to host B.

As if things weren't bad enough yet, the next, now deliberate, step by the network engineers is to openly denounce and reject end-to-end transparency on the pretext that it allegedly harms Internet access security and privacy. As soon as this thesis is accepted as an unchallenged truth, NAT boxes and proxies come to be regarded as ultimate network guardians and the IT market trend follows. Let's see if this now popular belief has any real basis.

First of all, we need to realize that *the issues of address space conservation, network security, and Internet access privacy must be considered and addressed individually* because they are completely different and each of them has its own set of specific requirements and rules. Simply put, you have no chance of hitting those three birds with a single NAT/proxy stone although you can wallow in the vendor-supported illusion you did. No doubt, NAT and proxies can conserve IP addresses, but that's about it. Attackers from the public Internet can, and every day do, easily infiltrate private networks, for example, by infecting a host on the target network through a malicious email message sent to the host's gullible or careless user.[3] Once infected, it is the victim host that initiates a connection to the malicious command-and-control center, and the NAT or proxy winds up facilitating rather than

[2] As we know, the Internet is a confederation of networks, where an individual network usually corresponds to a single administrative domain.

[3] At the time of writing, the most notorious attack of this kind was that on RSA Security [8].

preventing the attack because, if not for the NAT/proxy, abusing the victim host would have been impossible due to no global IP address on it. Neither does IP masquerading really help to maintain Internet access privacy and anonymity. The user's identity can be revealed as effectively, for one thing, by application layer data such as email headers and HTTP request fields as well as characteristic values in Layer 3 and 4 headers if correlated with respect to time.

A trivial example of a host identity leak through NAT is when a user host behind a NAT is just incrementing its IPv4 **ID** or ephemeral TCP port number. In this case, a remote observer can easily guess which of the IP packets coming out of the NAT originated from the same host although there can be multiple hosts masqueraded to the same public IP address. References to more sophisticated reconnaissance techniques targeting NAT'ed networks can be found in [10].

The misconception that NAT is good for network security often develops into a belief that NAT is a completely trouble-free way to protect a private network in a fire-and-forget manner. The proponents of this view tend to overindulge in forgetting and so they can never remember how much effort it takes to overcome NAT's unfriendliness to many existing protocols. To name just a few examples of importance, NAT can be an issue to IPsec [9, 11], FTP, and SIP. There already are well-established ways to work around this problem such as NAT-T for IPsec and STUN for SIP, but they, too, take human time to develop, implement, and support them.

Other network applications requiring a known, or at least fixed, local endpoint address can go to great lengths to trick the NAT middleboxes into keeping their public IP address and TCP or UDP port constant for an extended period of time. This class of tricks is known as **UNilateral Self-Address Fixing** (**UNSAF**) [12]. In turn, UNSAF can introduce even more security issues in a poorly protected network relying on a NAT box as its only gatekeeper because an attacker can as well take advantage of a fixed target address.

The practical difficulties caused by NAT are just the visible tip of the iceberg. The use of NAT has much deeper consequences for the Internet architecture, as discussed in [13].

Now that we see why NAT and proxies are inherently unable to provide comprehensive protection to a private network, let's move on to scrutinizing the basic proposition that end-to-end transparency is insecure. Can a fundamental principle be insecure? Granted, it can, but only if it immediately leads to dangerous practice, which hardly applies to end-to-end transparency. A fine point to grasp here is that end-to-end transparency does not mean full access with hosts allowed to exchange data in a totally uncontrolled manner. Global addressing only lays the technical groundwork for free and unhindered communication between hosts, but the actual decision whether a particular conversation should be allowed is up to the network admin. That is to say, end-to-end transparency makes direct communication between hosts always possible in principle, but it can still be blocked by administrative means. However, should we choose to ditch end-to-end transparency, there will inevitably be cases where direct communication between hosts is welcome but technically impossible. Any engineer knows how embarrassing it can be to end up in such a deadlock. It would indicate a major flaw in the architecture.

Of course, NAT and proxies are not the only enemies of the end-to-end transparency of the Internet. A more comprehensive blacklist can be found in [7].

The end-to-end transparency of the Internet can be restored only by increasing the number of global IP addresses. There is no other way around it—at least as long as we adhere to TCP/IP. The total number of IPv4 addresses was limited because the IPv4 address was essentially a fixed-length bit string with 32 bits in it, having as few as 2^{32} distinct values. We are not going to give up on binary representation of data any time soon, so all we have to do is increase the number of bits in the IP address. But how drastic a change will this address extension cause in TCP/IP?

Let's first try a moderate approach minimizing changes. Can we extend the IP address without breaking the existing IPv4 protocol and, in particular, its header format? Let's see. The first step could be to store the new, extended, addresses in IPv4 options designed for the purpose. The new IP addresses would pose no problem to ARP because the address lengths are explicitly specified in ARP messages. At the same time, ICMP is sensitive to the IP address format since some ICMP message types include IPv4 addresses in their body. This is a serious obstacle to our moderate proposal because ICMP is an integral part of the IPv4 stack. In addition, a fixed address offset in the IPv4 header makes hardware-assisted forwarding easier and faster whereas option-based addresses would be a hindrance to it. Finally, protocol options are called so because implementations may opt to support or ignore them, and this is the main reason why it would be a poor choice to put the new long addresses in IPv4 options. On the other hand, the fixed part of the IPv4 header only provides room for 32-bit addresses. So our moderate proposal fails: We are going to need a new IP header format along with extended addresses, at least to accommodate the latter.

Now that we are ready to say good-bye to the IPv4 header format, we can't but yield to the opposite temptation: to redesign the IP protocol from the ground up in order to free it from the historic shortcomings as well as to equip it with cool new features. This is going to be our main goal for the rest of this book.

How different will the new protocol be from IPv4 as we know it? Given our revolutionary spirit, the two protocols will be likely to have little in common as far as their wire data formats are concerned. If so, we need to provide, first of all, for a reliable way to tell old-style IP packets from new-style ones. Let's therefore keep in place the very first nibble of the IP header (**Version**) as shown in Fig. 1.1 so that we are free to rebuild everything else. The **Version** value in an IPv4 packet is, as one would expect, 4 and for the new protocol we will need a different version value. If we stick to this scheme, the new protocol will count as just another version of IP, and the TCP/IP stacks out there will be able to tell which protocol an incoming packet complies with by simply checking its first nibble.

Let's pretend that we went to *IANA* and requested a new value from the IP version registry [14]; *IANA* had to skip version 5 because it was already taken and handed us version 6. So now we can give name to our future protocol. It will be known as "IP version 6", or **IPv6** for short.

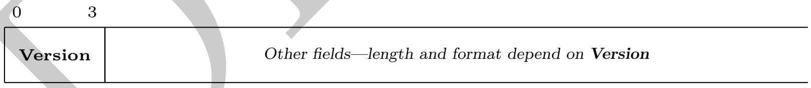


Figure 1.1: Version-agnostic IP header format

> In the above specification, we failed to mention an essential detail. What do you think it was? (Hint: *Endianness. The bit and byte order adopted by TCP/IP is MSB first, also known as "network order". Therefore* ***Version*** *occupies the most significant bits of a packet's first byte. For instance, most IPv4 packets have their first byte set to a value of* $(4 \times 2^4) + 5 = 69$ *because their version is* 4 *and they have no IPv4 options in them, so their header length in 4-bit words is* 5.)

Later on, in Section 4.1, we will comment on whether IPv4 and IPv6 are different protocols or just different versions of the same Layer 3 protocol. Generally, this is a sensitive topic prone to holy wars, so we will try to remain neutral and prove neither point but merely exercise our comprehension of the network protocol design basics.

Still, we will encounter quite a few architectural aspects common to IPv4 and IPv6. When speaking of them, we will use a version-agnostic designation: **IP**.

Chapter 2

IPv6 Address

2.1 The IPv6 Address Length Issue

Our immediate goal is an extended IP address space. Therefore it will be natural to start our "IP revolution" at the network node address, and the first question about the new IP address can be put as straightforwardly as this: How many bits will be needed in the new address to make future revolutions unnecessary? A number of different answers to it have been offered over the years, but most of them fall in one of the following three categories:

1. "Don't Bother Guessing": The new IP address must have a variable length to be able to cope with any rate of the growth of the Internet.
2. "Ask Mother Nature": Let's pick some totally off-the-scale number such as an estimate of the atomic particle count in the Universe and conjecture that people can never need more IP addresses than that.
3. "Qualities Before Quantities": The new IP address must be long enough to open up new exciting opportunities in the protocol design area.

The first way to tackle this problem seems by far the best—in theory. Indeed, a variable address length with a reasonable upper limit on it would sustain address space growth for millennia. Furthermore, this way is anything but new: The ISO CLNP protocol has relied on it for years. On the other hand, a variable-length address poses a challenge to fast handling by the switching hardware. In addition, the header format would no longer be fixed, its fields floating depending on the lengths of the addresses, and this would make efficient hardware switching even more difficult. This problem is further exacerbated by the simple fact that extra complexity entails cost increase. If the IP address length were to become variable, the cost of all IP packet processing software and hardware would inevitably go up. Considering that literally every home appliance will soon have an embedded TCP/IP stack in it, we must not take this aspect lightly, for it can doom the whole project. All in all we have to admit that a variable-length IP address was not a really great idea.

It is interesting to note nevertheless that an early proposal for a "next-generation IP" known as **TUBA** would simply stack TCP and UDP on top of ISO CLNP [15]. A certain elegancy of that proposal is hard to deny; but even the TUBA developers themselves voiced concerns about the extra complexity resulting from CLNP's great flexibility [15, Section 6.5]. If TUBA had ever succeeded, now we would be "designing" IPv9 rather than IPv6 [14].

The second approach (Ask Mother Nature) seems so attractive to many because it appears to immediately provide a rough estimate of the number of addresses needed and so all the protocol designer needs to do is take the binary logarithm of that number, round it up to a nice word boundary, and call it the answer. But are there any real grounds to draw a link between some weird particle count and the required IP address space size? True, even the human ability to waste things has a limit hardly higher than the astronomical numbers quoted by the proponents of this approach, so it can't be seriously argued that their guesstimates will be totally incorrect. However, the actual problem with this method hides in its deceptive simplicity, which can divert us from valid technical criteria relevant to TCP/IP rather than to cosmology. So we had better resist this lure and take a harder track so as to arrive at a more grounded decision in the end and get a better understanding of how IP works in the real world by the way.

A less obvious trap here is that a sloppy way of thought unsupported by rigorous and relevant logic can too easily become a habit and lead to a downright blunder. When blindly guessing the required number of addresses, the "naturalists" often fail to validate their own basic assumptions. An extreme example of this kind of flaw can be found in another early proposal for next-generation IP, where its authors used the overall number of online storage bytes to estimate the required address space size, assuming a single online resource would never be over 4GB in size [16, Section 2.1]. We hope it is quite obvious that their real problem was not just in the accuracy of the online storage size forecast but in the very premise that the target address space size was mysteriously linked with the number of bytes to appear online.

We still can't help satisfying our curiosity and seeing how great the real astronomical numbers are. Believe it or not, but their order of magnitude was known to a scientist as ancient as Archimedes. Once Archimedes had to demonstrate the power of science to King Gelo in order to secure state funding for his research. To impress the king, Archimedes developed a pretty solid theory [17] that enabled him to reckon the grains of sand in the Universe at 10^{63}. The modern astrophysical theories speak of atoms rather than of grains of sand and conclude that the Universe, depending on the particular model applied, has 10^{71}–10^{87} atoms in it. Assuming the average sand grain diameter of 300 μm, its density of 2.4 g/cm^3 (quartz), and its mean atomic weight of approx. 20 $a.u.$ (SiO_2), such an average grain of sand has about 10^{18} atoms in it. Then Archimedes's Universe contained 10^{81} atoms, which was a fairly good estimate even by the modern standards. What has just been proven here is that the power of scientific imagination has been effectively constant since the ancient times.

Now that a major problem has been exposed in the Ask Mother Nature method, we can look behind its false curtain and see what it obscured from us. Its lame reason effectively tried to make us at peace with a flawed idea that all IPv6 addresses would be used up someday, too, and we could do no better than just put off this dreaded event until as far in the future as possible by providing a completely off-the-scale amount of addresses. In other words, the IPv6 address space was still regarded by the "naturalists" as a formless sand pile to be spent a grain at a time, or by bucket, or with a backhoe; so the larger it would be, the better. But a real address space is never amorphous: It always has a certain structure, and now is the right moment for us to focus on shaping one for IPv6 because it will have a direct impact on the new address space longevity and usability. We will be rewarded in return with a better-grounded figure for the required address space size.

As we can see now, the problem with the Ask Mother Nature approach to IPv6 address space size assessment was an architectural one. It would hardly have resulted in premature depletion of the IPv6 address space, but it could have sidetracked us and made us miss some points of paramount importance.

To start with, let's realize that the IPv6 address will remain binary, i.e., it will still be a bit string, and it is handy to distribute parts of a binary address space by binary prefixes rather than by individual addresses or arbitrary address ranges. We know this procedure really well in IPv4. An entity in the address allocation hierarchy, say, an LIR, receives a certain binary prefix from the upstream (RIR), appends a few bits to it and passes the resulting extended prefix downstream (to an end user). Let's assume for simplicity that the entity always appends the same number of bits n to its master prefix to derive prefixes for its subordinates. Then it will be able to carve up a single master prefix into up to 2^n longer prefixes representing smaller address blocks. For example, given the IPv4 prefix *198.51.100.128/25* by the upstream authority (e.g., an LIR) and using $n = 2$, one can derive four */27* prefixes to assign downstream (e.g., to end users) as follows: *198.51.100.128/27*, *198.51.100.160/27*, *198.51.100.192/27*, and *198.51.100.224/27*.

What should the entity do when its current master prefix runs out? Apparently, it has to request another master prefix from the upstream. This new prefix inevitably gets used up, too, and yet another prefix is requested in order to sustain the network growth. As time passes, each entity in the hierarchy ends up having multiple prefixes that cannot be aggregated because they were derived from different master prefixes upstream. For instance, *198.51.100.128/25* won't aggregate with *203.0.113.0/25* while it could have aggregated with *198.51.100.0/25* into the single prefix *198.51.100.0/24*, had the other half not been assigned to a completely different entity. This is how the IPv4 address space got fragmented heavily.

> A theoretical alternative to maintaining multiple master prefixes is to hand the old prefix back to the upstream authority and request a shorter one instead in order to increase the available number of bits n. But in practice, this way is only feasible to leaf entities in the hierarchy, which have kept all their addresses to themselves and never allocated addresses to anybody else; otherwise the entity would have to *revoke* all allocations it has made to its subordinates. The political aftermath of such a draconic step can be easily imagined. But even at the hierarchy bottom, replacing a master prefix means network renumbering, which is a doable yet extremely painful procedure as it involves changing the addresses of all nodes in the network.

Why is address space fragmentation considered a bad thing? Firstly, it means more administrative overhead as each entity has to keep an account of multiple master prefixes and multiple allocations to each of its subordinates although in a perfect world each entity would need no more than one master prefix of appropriate size from the upstream. Secondly, the number of routes in a network is proportional to that of its non-aggregatable prefixes. For instance, in January, 2010, the Internet core known as the default free zone (**DFZ**) had about $30,000$ autonomous systems and $150,000$ IPv4 routes in it [18], a fivefold difference being mostly due to IPv4 address space fragmentation.

To avoid this problem in the future, *the IPv6 address space must be large enough to be inexhaustible at each distribution tier* or, simply put, a single properly sized IPv6 prefix should be sufficient for any entity in the hierarchy. Note well that we aren't going to prohibit allocating multiple master prefixes to a single entity: There will be valid cases where multiple allocations are useful and technically justified. However, we have got to put an end to the current IPv4 case where the dominant reason for doing multiple allocations is chronic address shortage at all tiers: The users are pestering their respective ISPs to get ever more addresses, the ISPs are soliciting new address blocks from the RIRs all the time, and the RIRs cannot but keep demanding new prefixes from *IANA*. Needless to say, this pyramid of need can't last for long, and providing enough room in a single prefix can be the only remedy for it possible.

> At the same time, it needs to be stressed from the outset that we do not encourage squandering the IPv6 address space, however large it be, for it will only stay inexhaustible if used reasonably and thoughtfully [19, 20].

To estimate the number of addresses required so, we need a certain model of the address allocation hierarchy. Now we are going to build one for our purposes. We will start from the very bottom of the hierarchy and define the base tier that can't be subdivided any further. A fundamental TCP/IP principle we aren't going to challenge holds that the tier in question is populated by links and the prefixes allocated to them are subnets. In other words, we are about to assume for the moment that a unicast IPv6 address, too, will consist of a subnet prefix and a host ID within the subnet.

While a single link can have multiple subnet prefixes assigned to it, it is dictated by another fundamental principle of TCP/IP that *no IP subnet can ever span multiple links* [21, Section 2.1][22]. This is a basic postulate many aspects of TCP/IP build on. Therefore the maximum number of addresses in one subnet can be estimated by how many nodes there can be on one link. Keeping in mind the modern Layer 2 technology trends, it can be reasonable to say that a single link can connect as many nodes as there are MAC addresses available. Today the traditional 48-bit IEEE 802 MAC addresses are gradually giving way to the 64-bit ones known as EUI-64 [23]. So, at least theoretically, a link with EUI-64 addressing can connect up to 2^{63} nodes. Based on this speculation, let's round the number up to the next full byte and make a tentative allocation of as many as 64 bits, or 8 bytes, of the unicast IPv6 address to the host ID.

Following the IEEE 802 practice, one EUI-64 bit, denoted as **I/G** (individual/group) or just **g**, is taken to distinguish link-layer unicast and multicast addresses. Hence the maximum of 2^{63} EUI-64 nodes on a link, not 2^{64} of them.

Alright, here we have a host ID that can last for a good while. And what can we figure out regarding the subnet prefix? For our choice of its length to be as reasonable, we need to go up the address distribution hierarchy and examine its tiers above the link tier. Unlike the latter, which is based on the fundamental notion of a network link, the upper tiers have more to do with administration than with technology, so we will have to accept them as they have developed. According to the current practice [24], there are four well-known tiers above the basic link tier:

1. End user: a person or organization
2. Local Internet Registry (LIR) held by an ISP
3. Regional Internet Registry (RIR)
4. *IANA*

Note well that a modern end user can no longer make do with one link, let alone a single IP address, even if it is a mere person. It goes without saying that an organization needs multiple IP subnets for its network. At the same time, a contemporary personal user, too, operates a whole network of desktops, laptops, handhelds, mobiles, cameras, coffee-makers, and toasters, if often unaware of that. It will be realized soon that personal networks cannot survive with a flat, LAN-like, structure having no security zones in it. How would you like this horror scenario for a near future: On a sunny Sunday, leisurely citizens witness a funny scene in the town park and, of course, they cannot resist capturing it with their mobile phone cameras. They are too amused to notice a viral QR-code a digital terrorist has put in the background. Same evening the virus spreads through personal networks and attacks vulnerable coffee-makers. Cut to the gloomy Monday morning. The victims reach for the hot steaming brew they are craving so badly, but only to get a mugful of lukewarm slops. The evening newspapers tell stories of a mysterious suicide wave that hit the middle-aged patresfamilias of the town... But the tragedy could have been prevented, had the mobile handsets and the coffee-makers been firewalled apart.

It was rather generous of us to reserve 64 bits for the node ID, and now we are tempted to keep spending IPv6 address bits in chunks of the same length. If we do so, the IPv6 address will wind up having 320 bits in it because there are five tiers in our model: a link, a user, an LIR, an RIR, and *IANA*. There can be no doubt that an address space so immense will last for ages, but is it going to be within the capabilities of the present-day computing systems, here and now? E.g., CPU-based systems can handle addresses as basic units only as long as they fit in the CPU registers, and if they don't, performance will be degraded. ASIC-based systems such as hardware router boards, too, have a certain limit to their switching matrix bit-width. And even if the circuit designers manage to come up with a system of sufficient capacity, it will inevitably be more complex and therefore expensive and, in addition, it will draw more power and produce more heat than a competing system based on a more balanced design of the address space trading off some of its capacity for practicality. Today fully 64-bit systems are the norm and 128-bit elements have started to appear while true 256-bit components still seem to exist only on the whiteboards. So let's check if we can make do with as few as 128 bits in total, thus leaving 64 bits to the four administrative tiers.

> We should also keep embedded systems in mind. They are still 32-bit or even 16-bit because that makes them more compact and power-efficient, but even so they don't lag far behind their big brothers with respect to support of TCP/IP networking! However, having to handle overlong addresses can stretch their limited RAM capacity because the longer the addresses are, the larger the network stack memory footprint has to be due to bulkier code and larger data structures.

Let's assume for the beginning that the 64 subnet prefix bits were evenly divided between the four tiers; that is, an end user would have up to 2^{16} subnets with 2^{64} nodes in each; an ISP LIR would be able to connect up to 2^{16} customers; an RIR would be able to support up to 2^{16} LIRs; and lastly, *IANA* would get 2^{16} top prefixes for architectural purposes as well as for allocation to RIRs.

Would 16 bits be enough for one entity? To put it into perspective, we can recall what the IPv4 practice was in this respect. Only largest IPv4 networks could afford 16 bits of what was generally known as a subnet ID. For example, to get that many subnet ID bits, provided that all subnets were */24*, an end user would need a public, globally routable, */8* (what old-timers knew as "a class A network"), each */24* subnet still handling fewer than 2^8 IPv4 nodes. And now 16 bits can be the basic minimum an IPv6 end user gets for subnetting, each subnet having up to 2^{64} nodes in it. As for the LIRs, prior to the latest IPv4 address restrictions in place, a new LIR would immediately get a */19*, which contained only 2^{13} *individual IPv4 addresses*, or 2^5 */24* subnets. So, even if the IPv6 subnet prefix breakup were as naive as we just assumed, an LIR would get thrice as many *bits*, not mere addresses, to work with in comparison with what it used to have in IPv4. All in all 16 bits *might* be enough for everybody.

In reality, the actual number of direct subordinates will grow as we go *down* the hierarchy. Indeed, *IANA* supports just a few RIRs, each covering a large area such as a continent or a close group of them. Each RIR, in its turn, handles hundreds or even thousands of LIRs, and a large ISP LIR can provide Internet services to millions of end users. It will be natural to allow for this uneven scale in our tentative IPv6 subnet prefix breakup by varying its portion length depending on the tier, for instance, as follows:

- 8 bits to *IANA*, allowing for up to 256 RIR allocations and well-known prefixes.

- 16 bits to an RIR, providing room for about 65 thousand LIRs.

- 24 bits to an LIR, enabling it to connect up to 16 million end users.

- 16 bits to each end user, for them to have up to 65 thousand subnets in their network.

When it ever comes to everyday practice, this breakup won't need to be fixed exactly like that because prefix-based address allocation has no problem with having the length of each prefix tailored to the requirements of its recipient in order to provide them with as many available bits after the prefix as they need [78, Section 5]. E.g., a small LIR can be happy with as few as 16 bits in its authority window while a big corporate user can request an aggregatable series of standard-size blocks from its LIR to get as many as 24 subnet ID bits, thus making up an impressive */40* block. But that will be the future and now we are still in the design phase and need to work out the mere IPv6 address length. So the above ballpark layout will serve for our purposes, an average IPv6 end user tentatively getting a */48* prefix and having 16 extra bits available for subnetting.

There have already been IPv6 allocations by an RIR to a big LIR as large as */19*, thus resulting in the LIR getting as many as 29 bits in its authority window [19, Section 2.4.1]—of course, not counting the bits the LIR's subordinates are going to get from it under their respective master prefixes. In our simple model, that LIR can connect up to 2^{29} end users, each still getting a fair share of the IPv6 address space.

There is a well-known trick for an address allocating authority to be able to extend a block already allocated by merging it with the adjacent aggregatable block [25, 26]. The challenge here is to ensure that the adjacent block stays free as long as possible. This can be done by introducing 1's in the most significant bits rather than in the least significant bits of the prefix when counting. For example, if you have 4 bits to work with, you will be allocating their consecutive values as $0, 8, 4, 12, 2, 10, 6, 14 \ldots$ instead of the natural series of $0, 1, 2, 3 \ldots$ Thanks to this procedure, you can allocate up to $1/2^N$ of all prefixes and still be sure that each prefix allocated has $2^N - 1$ adjacent prefixes free and all 2^N contiguous prefixes have N least significant bits set to zero, providing for their aggregation into a single prefix when needed. For example, as long as only 1/4 or less of available prefixes have been handed out, you can extend each block of addresses allocated two- or fourfold. An in-depth analysis of this trick's binary arithmetic is left as an exercise to the reader. (Hint: *Convert the integers* $0, 1, 2, 3 \ldots$ *to a binary notation of fixed bit-width and then write each of them back to front.*)

The bottom line of our speculation is that the 64-bit subnet prefix has enough room in it for flexible hierarchical allocation of IPv6 unicast addresses. Another 64 bits were reserved for something we tentatively referred to as a node ID within the subnet, so the two fields will occupy 128 bits in total. We will now stop short of defining the actual names and meanings of those fields in the IPv6 unicast address because all we wanted to figure out first was just the length required for the generic IPv6 address. Assuming other address types won't need to be longer than the unicast address, we can come up with our first specification: *The IPv6 address will be a fixed-length bit string with 128 bits in it* [21] (Fig. 2.1).

0
127
node address

Figure 2.1: Generic format of IPv6 address

We postpone the numeric interpretation of the IPv6 address until Section 2.2. Nevertheless, it should be realized early on that the bit string known as the IPv6 address is ordered, with its beginning and end clearly defined. If you put it back to front, you will get a different IPv6 address unless the bit string sample on hand is symmetrical with respect to reversion.

We are still tempted to speculate that, had IPv6 been born in 2010, its address would have extended to 256 bits or more. First, that would have created more freedom with respect to the number of address distribution tiers IPv6 could afford. Second, IPv6 address conservation concerns [20] wouldn't have had to be raised so soon. At the same time, a sufficient but limited number of prefixes will encourage the network engineers to think twice when working out a network's address plan, and thinking is always good for people. An abundance of digital resources can never be an excuse for spending them in a reckless manner.

Here's an interesting historical detail to reflect on: On the 1st of January, 1983, the Internet of the time concluded its transition from NCP to TCP/IP in order to solve the same problem of extending its address space and, ironically, the new address was just four times as long, 4 bytes in IPv4 versus 1 byte in NCP. Does it ring a bell? Nothing is truly new in this world, IPv6 not excepted!

Let's estimate how many IPv6 addresses will have been allocated when the HD ratio (see Chapter 1) reaches a threshold of 0.8, touching off alarm bells for the IPv6 address space utilization [4, Section 4]. In this calculation, using binary logarithms will be handy. The binary logarithm of the number in question equals to $0.8 \times 128 \approx 102$. In other words, there will be grounds for concern *26 binary orders of magnitude* before the complete exhaustion of the address space. It is in the essence of the hierarchical distribution paradox: Address space exhaustion makes itself felt well before we are anywhere near the theoretical limit.

2.2 Text Representation of an IPv6 Address

Now that the IPv6 address has been cast in its basic form of a bit string, it is time we provided an example of a real IPv6 address in this printed page; but how shall we spell it? As a sequence of zeros and ones? Well, we could do so once for illustration purposes, but having to do that all the time would be a gross inconvenience. So let's not until we have a better notation for IPv6 addresses in order. To see how the notation can be improved, let's tell what is wrong with the bit string spelling. First, it is extremely verbose and redundant with respect to human writing since it is based on a 2-symbol alphabet: 128 characters per address is way too many for a human, precluding its error-free handling. By introducing additional symbols to use, we can make the thing more compact. Second, different human writing systems employ different writing directions. Were an IPv6 address to be written down as a mere bit string, different nations would have full right to write the same address differently. To avoid the babel, the text notation of IPv6 addresses has to comply with a mini-protocol shared by all parties concerned.

So our immediate task is to come up with a compact and unambiguous text representation for IPv6 addresses. Its primary purpose will be to let people write such addresses down, read them, and, to some degree, remember them. However, network addresses aren't only written down on paper: They are entered into computer systems as well. For example, address entry is what manual address configuration boils down to. In addition, addresses are read off the screen and printed out. For a computer system to be able to process a text notation of an address, the notation has to be representable in a text encoding supported by the system. We don't need to be concerned about the actual text encoding rules now, but whatever characters the address notation will use have to be present in the encoding repertoire. For this reason, it would be wise to limit the notation to a subset of symbols shared by all existing encodings including exotic or historic ones such as EBCDIC. Fortunately, such a subset exists and consists of Latin alphabet letters (irrespective of their case), decimal digits, and basic punctuation marks.

> Which punctuation marks can be regarded as basic? Generally, it is safe to assume they are those present in the ASCII repertoire because ASCII still remains the basic encoding of the Internet [28] while UTF-8 is yet to take this role over [29]. An even better choice can be the punctuation characters shared by ASCII and EBCDIC.

In IPv4, the same problem was solved with the well-known "dotted decimal” notation of addresses. Let’s recall it in some depth:

- What is given on input is an IPv4 address as an *ordered* 32-bit string.
- The first step is segmenting the address in 8-bit chunks (bytes) so that the relative order of bits is preserved.
- Then each byte is interpreted as an unsigned binary integer in network order, where the first bit is the most significant one,[1] and its value is written down as a decimal number.
- Lastly, the byte values are listed in order so that the first and most significant byte of the address is the leftmost and the last one is the rightmost. The four byte values are separated by periods, aka “dots”.

Quite surprisingly, the wide-spread tradition of writing numbers in a positional system so that their most significant digit ends up the rightmost and the least significant one the leftmost shows no direct connection with the direction of writing. For instance, it equally applies to Arabic writing, which is right-to-left. Apparently this tradition predates the writing systems in modern use and goes back to the single mysterious source many a culture borrowed the idea of positional number notation from.

The discussion of the applicability of this kind of notation to vertical writing systems is left as an exercise to the reader.

[1] A common shorthand for that being, “MSB first”.

An issue with dotted decimal notation is that it didn't specify from the outset whether it was OK for the byte values to have leading zeros in them. Of course, if the byte values were mere arithmetical numbers, they would be free to contain leading zeros because leading zeros don't change the numeric value in any positional notation.

However, addresses in dotted decimal notation have to be processed by computers and that is where a real problem starts. In computing, the interpretation of a number can change if it has a leading zero in it just because the leading zero is meant to convey extra information to the computer system beyond basic arithmetic.

The C language and its descendants, in which a leading zero flags an octal number (i.e., base 8), are a case in point. Extending beyond the C language proper, this interpretation has made it into the standard C library (see *strtol*) and from there it may have crept into applications, more due to a sloppy programming style than to the C library feature itself as the API still supports nailing the radix down in the code rather than having it guessed from the input format at run time.

For instance, the *inet(3)* IPv4 address parsing functions from the *BSD libc* tolerate any radix supported by the C language: 8, 10, 16 [81]. Base 8 or 16 can be selected with a leading `0` or `0x`, respectively.

To put an end to this confusion, the latest ruling included in the URI specification has completely prohibited leading zeros in IPv4 address octets [27, Section 3.2.2].

Lesson learned, in the IPv6 address notation we will do the right thing and specify the exact meaning of leading zeros so that code developers can handle them right.

What can be improved in dotted decimal notation? First, decimal notation still has a high redundancy, which can be further reduced by switching to hexadecimal. Indeed, a number rewritten in hex can be $\log_{10} 16 \approx 1.2$ times as compact. It isn't a huge win but it still can matter for an address as long as 128 bits.

Second, the decimal notation of an IPv4 address has been a bit of a pain to match with netmasks and packet hex dumps. Of course, converting decimal to binary and back is an excellent exercise for one's brain, but there is going to be too much of it in the IPv6 world unless we give up on decimal in favor of hex now.

Lastly, a hex number is very easy to split into bytes because each hex digit stands for a nibble and a pair of them make a complete byte. By contrast, leaving the dots out of an IPv4 address will be likely to result in a number completely different from the numeric value of the address; and conversely, the accepted notation of an IPv4 address can't be obtained by merely inserting dots into its decimal value.

For example, the IPv4 address *203.0.113.42*, if regarded as a 32-bit unsigned integer, is equal to 3405803818 decimal or `0xCB00712A` hex. Unsurprisingly, making out the individual byte values is straightforward only in the hex: `0xCD` is 203, `0x00` is 0, `0x71` is 113, and `0x2A` is 42.

So there is more than one good reason for the IPv6 address notation to use hex.

> Meanwhile, Australian IPv6 researchers experiment with an address notation base 85 [30].

The first problem for us to solve is as follows: Positional notation is used to represent numbers, so we need to clarify first of all what the numeric value of an IPv6 address is. As we know, a bit string can be regarded as an unsigned integer number provided that it is known which of its ends is the more significant one. TCP/IP leaves no options to us in this regard: The bit order of choice is MSB first. Thus each IPv6 address is, in a sense, just a pretty long number base two.[2] It is guaranteed by the arithmetic of fixed-width positional notation that the mapping between IPv6 addresses and their numeric values is one-to-one, so that no two different addresses can have the same value while different values never map to the same address.

> At the same time, a variable-length address system can have different addresses map to the same numeric value due to the leading zeros issue. For example, this will be the case in a system where the bitwise addresses `1` and `01` are valid and different.

The next step will take us from the abstract numeric value of an IPv6 address to its conventional hex notation. A 128-bit string takes 32 hex digits, including leading zeros if any, to express its value. For convenience, let neutral separator characters be inserted between groups of hex digits in about the same manner as phone numbers become more readable if written with dots, hyphens, or spaces in them. For our notation to be more consistent than that of phone numbers, each group will have a fixed length of four hex digits and the only separator character allowed will be the colon. This will also make IPv6 addresses clearly distinct from their dot-separated IPv4 counterparts wherever they appear alongside in text.

[2] Of course, the same applied to IPv4 addresses.

Using the colon as the group separator may not seem the best choice possible because the colon already has several meanings in related formats. First, it separates the host from the port in the URI format. Consequently, a bare IPv6 address can no longer be specified for the host in a URI: It has to be enclosed in brackets to avoid ambiguity [27, Section 3.2.2].[a] (The same applies to any future version IP address just in case.) Second, the colon is an alternative group separator in the text notation of MAC addresses.[b] But, on the other hand, there are just a few suitable punctuation characters available in the ASCII repertoire. First of all, a good candidate must be neutral with respect to the *Unix* shell because IP addresses are often supplied as arguments to various network tools such as *ping*, *telnet*, and *tcpdump*, to name a few. Were the new addresses to require escaping or quoting, the Principle of Least Astonishment (POLA) would be downright violated.

[a] Note how the Internet email developers foresaw and handled this issue many years ahead of IPv6: The domain literal format mandated enclosing host addresses in brackets from the very beginning and regardless of the Layer 3 protocol in use [32, Sections 3.3 and 6.2.3].

[b] IEEE 802 states that the standard separator in a regular MAC address is the hyphen while using the colon instead indicates a bit-reversed MAC address [31, Sections 3.1.2 and 3.1.8]. However, in practice the colon can be encountered in MAC addresses having the normal IEEE 802 bit order, that is, LSB first. For instance, that is the case in many *Unix*-like systems.

For example, the IPv6 address having this numeric value, in hex:

```
20010DB8000000000000000100006789
```

will be written as follows:

```
2001:0DB8:0000:0000:0000:0001:0000:6789
```

This was our first *real* example of an IPv6 address in text. It fully complies with the format and is ready for use in applications. However, its high redundancy still stands out, its text form having runs of zero digits in it. It isn't just this example: By our master plan from Section 2.1, the IPv6 address space will be inexhaustible at each distribution tier and hence quite sparse; so runs of zero bits will naturally appear in a typical IPv6 address. Can such runs of zero bits be compressed or eliminated when in text? Not unless we come up with a couple of optional formatting rules.

First, let each four-digit group be regarded as an individual 16-bit unsigned number. Once its bit length and order are fixed, its numeric value has an unambiguous one-to-one correspondence with its bitwise data. Then we are free to discard any leading zeros from the group because that will have no effect on its numeric value by the basic properties of positional notation. Therefore leading zeros in the groups of digits can be safely declared optional.

> At the same time, leading zeros are completely legitimate in a group of digits of an IPv6 address and don't change the group's meaning. Regardless of their presence or absence, the group is a hex number. That said, a group can't have an arbitrarily large number of leading zeros in it because its total length must not exceed four hex digits. This restriction is just to emphasize that a group always represents 16 bits of the IPv6 address.

With this simple amendment, the same IPv6 address we used in the above example can get a more compact form:

```
2001:DB8:0:0:0:1:0:6789
```

Nevertheless, there still is a run of as many as three zero groups in it. What if we leave them out completely? We are certain that, in total, there are to be eight groups of digits in an IPv6 address, and this makes working out the number of groups omitted really straightforward. Let's try this on our sample address:

```
2001:DB8::1:0:6789
```

As there are to be eight groups in the address but only five of them are present explicitly, three contiguous zero groups were omitted, their position clearly marked by the double colon.

If needed, an address can start with a double colon or be terminated in it rather than have it in the middle, meaning a run of zero groups at the beginning or the end of the address, respectively. For instance, the following two notations are equivalent—but distinct from *2001:DB8::1:0:6789*:

```
2001:DB8:1:0:6789::
2001:DB8:1:0:6789:0:0:0
```

And what if we omit two or more runs of zeros? Consider this example:

~~2001:DB8::1::6789~~

Let's try to expand it back to a full address. It can be easily seen that four groups of digits were left out in two runs, but we seem to have no information at all as to how the groups were distributed over the runs. Was it 1:3, 2:2, or 3:1? No idea. This ambiguity prevents us from restoring the full address. Due to it, an IPv6 address can never have more than one run of zeros contracted to a double colon in it.

On the other hand, we are free to put a double colon in place of any run of zeros in an address. It doesn't have to be the longest one because the address can still be expanded unambiguously. For example, our first address *2001:DB8::1:0:6789* can also be spelled as follows:

```
2001:DB8:0:0:0:1::6789
```

When this spelling needs to be expanded back to the full address, e.g., by the software, it won't be hard to figure out that just one zero group was left out as the other seven are there.

> The abbreviation rule we have just formulated will make it easier to memorize IPv6 addresses. If anybody complains to you that IPv6 addresses are a pain to remember, please take time to explain the double colon trick to them.

> Some parsers of the IPv4 dotted decimal format allowed for a shorthand notation of sorts, the *inet(3)* functions from the *BSD libc* being a notable example. For instance, *10.0.0.1* could be spelled as `10.1` and *192.168.0.1* as `192.168.1`. Those compaction rules were more complex than just leaving zero octets out—see the corresponding *man* page [81] for details. However, such extensions to the dotted decimal notation were non-standard and implementation-specific, unlike the *well-known* IPv6 notation rules.

> What is the maximum number of colon symbols a valid notation of an IPv6 address can contain? (Hint: *Eight. An example can be found in [21, Section 2.7.1].*)

In addition to complete 128-bit IPv6 addresses, a notation for IPv6 prefixes will be required in certain practical cases. E.g., prefixes will appear in routing table entries as well as represent blocks of addresses allocated for a specific purpose. In its essence, a prefix is just a bit string not longer than a complete address. By convention, such a bit string represents all addresses containing it in their most significant bits. Due to their close connection with addresses, it is handy to express prefixes using the same notation extended with the prefix length. For the prefix notation to be consistent across the IP versions, let's just reuse the well-known IPv4 CIDR notation [33, Section 3.1] and require that the prefix length be written in decimal after the address part and separated from it by a slash character.

Suppose the following 58-bit prefix is given, in binary:

```
0010000000000001000011011011100000000000000000001100110110
```

How shall we express it using the IPv6 text notation? First of all we need to make a complete IPv6 address of it so that we can put it before the slash separator. To do so, 70 zero bits need to be appended to the prefix *from the right*, for the prefix represents the most significant bits of an IPv6 address. Consequently, the hex value of the resulting address will be this:

```
20010DB80000CD800000000000000000
```

And now that we have this value, all we need to do is format it using the rules we just worked out, then append a slash character and the prefix length (58) as follows:

```
2001:0DB8:0000:CD80:0000:0000:0000:0000/58
```

This was a perfectly valid example of IPv6 prefix notation. However, it can still be compacted by eliminating extra zeros from its address part as follows:

```
2001:DB8:0:CD80::/58
```

Lastly, sometimes it can be handy to combine an IP address and its prefix of a certain length in a single notation. The primary use case of this is in assigning addresses to network interfaces. By definition, the prefix is already contained in the most significant bits of the address, so it will be sufficient to extend the address with the prefix length using the slash notation. For example, this entry:

```
2001:DB8:0:CD9F::123/58
```

stands for the address *2001:DB8:0:CD9F::123* in the context of the prefix *2001:DB8:0:CD80::/58.*

Verify that the address *2001:DB8:0:CD9F::123* actually contains the prefix *2001:DB8:0:CD80::/58.*

Table 2.1: Expanding IPv6 prefix notation to raw hex

Text notation	Prefix (hex)	Trailer (hex)
`2001:DB8:0:CD80::/60`	`20010DB80000CD8`	`00000000000000000`
`2001:DB8:0:CD8::/60`	`20010DB800000CD`	`80000000000000000`

Such an address-and-prefix notation can't always be told from a pure prefix one unless enough context is provided. Of course, more often than not an address will have non-zero bits after the embedded prefix whereas a pure prefix will have all of the extra bits set to zero. Nevertheless, the address whose trailing bits following the embedded prefix are just zero is equally valid.

A likely error in a prefix notation can be to leave out trailing zeros in the last significant group [21, Section 2.3]. For example, the following two notations *are not* equivalent:

```
2001:DB8:0:CD80::/60
2001:DB8:0:CD8::/60
```

To see why it is so, we need to go back down to bit level, that is, to unfold both text notations to their respective raw binary or hex forms and compare those as shown in Table 2.1.

> The more compact hex form is only applicable if the prefix length is a multiple of 4 because a hex digit substitutes for a group of four binary digits (bits). Otherwise the binary form has to be used so as to avoid confusion and information loss due to effective prefix length rounding.

These rules of IPv6 address and prefix notation are documented in [21, Sections 2.2 and 2.3]. What [21] fails to mention is that the text forms in question are case-insensitive, which is a common practice with respect to hex: The letter digits can be lower-case (`a`–`f`) as well as caps (`A`–`F`) and the cases can be freely mixed [34, Section 2.3].

> When reading [34], please keep in mind that it is only concerned with the *output* format of IPv6 addresses, i.e., how applications are to display them. The authors of that Standards Track RFC failed to lay enough stress on this "minor" detail and left [34, Section 4.3] effectively open to the misinterpretation that the caps (`A`—`F`) are now completely banned from IPv6 addresses. There will be little grounds for surprise if soon there are applications or libraries out there rejecting upper-case IPv6 addresses in their *input* on this basis.

There also exists a slightly different notation for IPv6 addresses in which the four least significant bytes are written like an IPv4 address. E.g., our first IPv6 address will be written as follows: *2001:DB8::1:0.0.103.137*. This notation can be handy in some IPv4-IPv6 migration and coexistence techniques based on embedding IPv4 addresses in the least significant bits of special-purpose IPv6 addresses.

2.3 IPv6 Address Types

Now we can start partitioning the IPv6 address space. What address types will be required? Based on our IPv4 experience, we can compile the following tentative list:

- Global unicast
- Private-use unicast
- Link-local unicast
- Multicast
- Loopback
- Unspecified

We will start working from the end of this list as it contains smaller and simpler address sets.

The unspecified address—it was *0.0.0.0* in IPv4—means that the address is unset, unknown, irrelevant. For example, in the Berkeley Sockets API an application would use the unspecified local address of a socket to request from the network stack that it pick a suitable local address itself. Quite obviously, the unspecified address can't be assigned to a network interface or appear on the wire in an IP packet header except in a few rare cases, each to be justified by a protocol. For instance, a host starting an automatic address configuration procedure will have little choice but to send its first packet with an unspecified source address and thus work around the chicken-and-egg problem that the host has to send a packet before it learns its first unicast address—we will discuss this matter in greater detail later on, in Section 5.4. At the same time, the destination address of a valid packet can never be left unspecified; otherwise it would be as meaningless as a blank address on a posted envelope. Despite its singularity, the unspecified address is going to play an important role in the API as well as in special cases of the protocol, and so its IPv6 value should be distinct and well-known; it needs to be nailed down from the outset. Let the unspecified IPv6 address, too, have a numeric value of zero:

```
0:0:0:0:0:0:0:0
```

Thanks to the compression rules from Section 2.2, that can be rewritten as terse as just a double colon:

```
::
```

The loopback address is required for TCP/IP applications to be able to work on a standalone computer with no real network interfaces. Moreover, it is handy to have a single well-known address always point to the local host because that really helps the increasingly common case when components of the same application need to talk to each other locally over TCP/IP as the IPC mechanism of choice. Instead of having to work out the local host address or to have the user configure it, such an application can just be using the same loopback address on any host. The loopback address is without doubt special in that it must never appear on the wire between hosts since it is meaningful to the local host only. In IPv4, a large block was initially reserved for loopback purposes, *127.0.0.0/8*; but ultimately just one loopback address, *127.0.0.1*, proved to be sufficient while the other uses enjoyed by that block were non-standard and often unrelated to the loopback. With this experience in mind, IPv6 will have a single well-known loopback address. For the sake of simplicity and brevity, it will be the address with a numeric value of 1:

```
::1
```

Multicast addresses deserve a unique well-known prefix so that they can be easily told from other address types by nothing else but their bits. That will facilitate multicast traffic handling as its rules are quite different from those for unicast traffic. For simplicity, let all IPv6 multicast addresses, and none but them, have a binary prefix of `1111 1111`. Using text notation, we can express it as follows:

```
FF00::/8
```

> Note well that it cannot be shortened to `FF::/8`. Please review Section 2.2 if you fail to see why.

Now on to unicast addresses. Our main goal is to provide the Internet with as many global unicast addresses as possible; for this sake, let's dedicate a finite number of smaller subsets of addresses to specific purposes and leave the rest of the vast IPv6 address space to global unicast addresses, thus maximising their number.

Link-local addresses already existed in IPv4. Back then they occupied the *169.254.0.0/16* block set aside by *IANA* for serverless autoconfiguration of hosts in a simple flat network [35]. Packets addressed from, or to, that block must have been confined to a single link and the hosts attached to it. We are likely to face the issue of serverless autoconfiguration in IPv6 again, so let's reserve the following block for link-local addresses:

```
FE80::/10
```

By our master plan, an IPv6 subnet prefix will always be 64 bits long. However, it is going to be a policy decision, not a protocol constraint. For this reason, allocating a shorter prefix to link-local addresses isn't in conflict with what we said earlier.

Private-use addresses were generally needed to deploy intranets, that is, private TCP/IP networks similar to the Internet by their technology and architecture but physically separate from it. Had it been known for sure that such an intranet would never have to get connected to the Internet, it could have used an arbitrary block of IPv4 addresses up to the entire *0.0.0.0/0* space. However, the reality was that sooner or later an intranet was to be connected to the Internet through a gateway of some sort, resulting in address ambiguity: The same address *A* could appear over here on the intranet as well as out there on the Internet, and if that was the case, the other intranet hosts could never be sure which peer *A* they sent packets to, the intranet friend or the Internet stranger. In the end, this sort of addressing fault was avoided once and for all by reserving certain address blocks for private use [79] and making sure they never appeared on the Internet. This approach wasn't perfect; we will discuss its drawbacks a few sections later, in Section 2.10. Nevertheless, it was effective in restoring address uniqueness as seen from a single intranet.

By now the term "intranet" has become largely obsolete inasmuch as there are few private TCP/IP networks left that are still truly standalone while the majority of them have connected to the Internet via a gateway masquerading their private-use addresses using NAT or application proxies. On these grounds, from now on we will stick to the more modern term **site** and give up on the aging intranet concept. A typical site is a company, or a unit in a larger company, whose network is managed by a single authority. Accordingly, instead of private-use addresses, in IPv6 we will be referring to **site-local addresses** that are only relevant to a particular site and are not to appear in the global Internet space. IPv6 site-local unicast addresses get the prefix adjacent to the link-local one:

```
FEC0::/10
```

> Ultimately, the practical drawbacks of site-local addresses outweighed their advantages and warranted their deprecation in IPv6 [36]. Later on, in Section 2.10, we are going to discuss what led to that decision and what the current alternative to site-local addresses is. Meanwhile, however odd it may sound, the site-local address concept will help us to grasp Section 2.4.

Finally, all other IPv6 addresses are global unicast. Well, almost all of them are.

> In fact, there is another couple of special cases, but we won't focus on them. Firstly, IPv6 addresses with a prefix of *::/96* are to embed IPv4 addresses in them, but only IPv4 global unicast addresses may be embedded this way [21, Section 2.5.5.1]; so the outer IPv6 address is, in a sense, global unicast too. Secondly, IPv6 addresses with a prefix of *::FFFF:0:0/96* can embed any IPv4 address in them [21, Section 2.5.5.2]. As of now, the former prefix, *::/96*, has fallen out of use and the only use of the latter, *::FFFF:0:0/96*, is to disguise IPv4 addresses in the API to a dual stack capable of both IPv4 and IPv6 so that IPv6-only applications can still speak to IPv4 hosts [39]. E.g., a TCP-based application no longer supporting IPv4 addresses can be tricked into connecting to *192.0.2.1* with this IPv6 address: *::FFFF:192.0.2.1*. See the note at the end of Section 2.2 for details on this special notation suited for embedding an IPv4 address.

> Currently, *2000::/3* is being allocated for regular use in the IPv6 Internet [41, 42]. The rest of IPv6 global unicast addresses are kept in reserve in case the current block runs out too early and the allocation policy needs to be revised.

IPv6 global unicast addresses, for the most part, are to be delegated to real Internet entities. However, we also need one address block for our immediate purposes, that is, for use in examples. The rationale behind such an allocation is pretty clear: Should an address from our examples end up by accident in the configuration of a live network, that still won't result in a global conflict. As we can recall from our IPv4 experience, there were three blocks set aside for this role, namely, *192.0.2.0/24*, *198.51.100.0/24*, and *203.0.113.0/24* [2, 43]. Back then multiple small blocks had to be allocated to make it easier to illustrate how different networks interact—without spending too much of the precious IPv4 address space. Now that we are in the IPv6 reality and have plenty of addresses, we can afford allocating just one larger block and leaving it up to the future authors to cut that block up as it suits them. It will be the block *2001:DB8::/32* [44]. As the reader might have noticed, we already started using it in our examples when speaking of IPv6 address notation.

> The block *192.0.2.0/24* was reserved for use in examples by the highest authority of *IANA*. The blocks *198.51.100.0/24*, *203.0.113.0/24*, and *2001:DB8::/32* were generously allocated by the Asia Pacific Regional Internet Registry, APNIC.

> As we know, IPv4 had an RFC document summarizing its special-use addresses [2]. There is a similar document available for IPv6 [45].

> If you google for "`2001:DB80::`", you will spot a number of online documents that use the non-standard prefix *2001:DB80::/32* in their examples, instead of the well-known *2001:DB8::/32*. Discuss whether it was a good choice on their authors' part and why.

2.4 IPv6 Address Scope and Zone

The link-local addresses we introduced in Section 2.3 are, in fact, just an application of a considerably more general and interesting concept we are about to explore. Before now, our experience was only with link-local addresses as they were implemented in IPv4, and that implementation was based on the assumption that a host had just one active network interface [35, Section 3]. That model provided for interconnecting a set of hosts with a LAN-style link and assigning a single prefix, *169.254.0.0/16*, to that link as a subnet, i.e., without further subdivision. Can we suggest a good reason to generalize this model to the case of multiple links?

To start with, consider the following network topology: There is a server with its interfaces connected to multiple LANs; workstations on those LANs

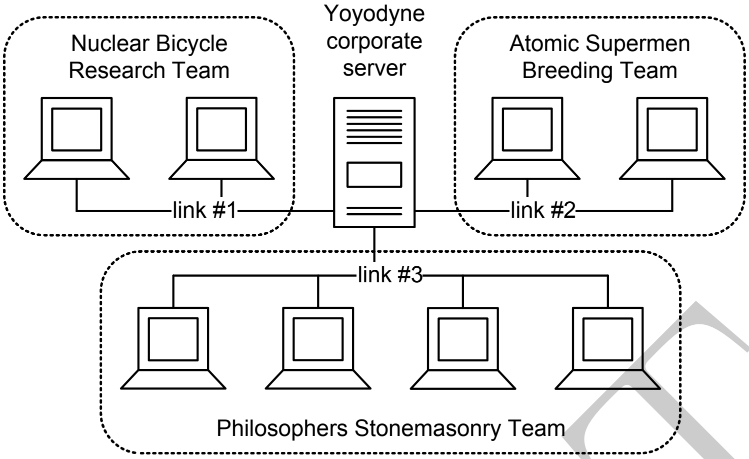

Nuclear Bicycle Research Team
Yoyodyne corporate server
Atomic Supermen Breeding Team
link #1
link #2
link #3
Philosophers Stonemasonry Team

Figure 2.2: *Yoyodyne* corporate network (imaginary)

speak to the server but not to each other, or at least not across the LAN boundaries. This topology can be illustrated by the imaginary *Yoyodyne* corporate network shown in Fig. 2.2. As long as each workstation speaks only to its link neighbors including the server, link-local addresses should be sufficient for this scenario. Each link can correspond, e.g., to a company division LAN, and using just link-local addresses will result in IP-layer isolation of those LANs by not routing IP packets between them.

However reasonable this scenario may look, there was no way to implement it in IPv4 because it would require an address from the same subnet, *169.254.0.0/16*, be assigned to each of the server's interfaces. The IPv4 stack would reject this configuration as invalid due to its ambiguity: Had it been accepted, the server would have had no criteria to choose a particular outgoing interface over the others when sending out a packet with a *169.254.x.y* destination address.

This limitation was taken for granted in IPv4, but now we can't help noticing that some important information was missing from IPv4 link-local addresses. Indeed, link-local subnets on different links should be independent of each other because each link-local address is to be confined to the respective link and, say, the address *169.254.5.6* on link #1 is distinct from the address *169.254.5.6* on link #2—at least in pure theory. However, the IPv4 stack lacked a mechanism to support this logic and so an IPv4 host was unable to tell apart link-local addresses or subnet prefixes on different links connected if they happened to have the same binary value [35, Section 3.2].

Now it is too late to fix IPv4 in this respect, but we are free to get it right in IPv6. The technical challenge here is to "teach" the IPv6 stack how

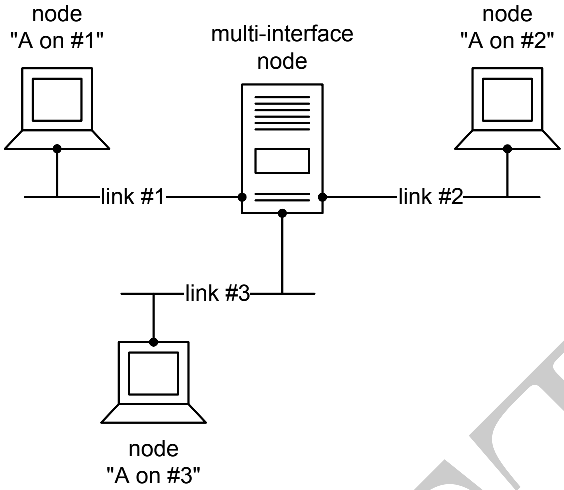


Figure 2.3: Resolving address *A*'s ambiguity by link index extension

to distinguish address *A* on link #1 from the numerically equal address *A* on link #2. To supply the information missing from its raw binary value, each link-local address needs to be extended with a suitable link index, e.g., as shown in Fig. 2.3. Then it will be possible to assign the same link-local prefix to different interfaces of one node, still causing no confusion to its IPv6 stack because now the stack can tell by the index which link a link-local address belongs to.

> A similar solution has been employed by common people as well as geographers to tell apart different towns of the same name providing they were founded on the banks of different rivers. Hence Stanford-on-Avon and Frankfurt am Main.

Attaching a link index to each link-local address is a great idea, but it will be truly brilliant if generalized beyond links. Let *each* IPv6 address have a certain **scope**, that is, a topological span within which the address is meaningful as a unique identifier of a network interface[3] [46, Section 4]. For instance, the scope of a link-local address is a link; that of a site-local address is a site; and the scope of a global address is a whole planet, or even a universe, depending on how high our ambitions soar. This concept lays the base of what is known as **IPv6 scoped address architecture** [46]. The only exception from it will be the unspecified address *::* owing to its special status: It has got no specific scope.

[3] Or of a set of interfaces, e.g., if it is a multicast address.

> Should you need to refresh your memory on the IPv6 address types, please review Section 2.3.

From this perspective, IPv6 link-local addresses are just one case the scoped addresses architecture is applied to. With this point in mind, let's revisit the example of a server at the center of a corporate network as shown in Fig. 2.2. Suppose each interface of the server has been assigned an IPv6 address of the same numeric value of *FE80::1/64* and hence of link-local scope—*FE80::/10* was reserved for IPv6 link-local addresses in Section 2.3, remember? According to our plan, the address *FE80::1* on link #1 is to be distinct from the address *FE80::1* on link #2 notwithstanding that their numeric values—and scopes—are equal. They are still different link-local addresses because they are on different links. To reflect this distinction in our technical lingo, we will say that link #1 is the **scope zone**, or simply the **zone**, of the address "*FE80::1* on #1". The same applies to any addresses in the subnet "*FE80::/64* on #1", their zone being just the same. In contrast to them, the addresses "*FE80::1* on #2" and, say, "*FE80::C0DE* on #2" come from a different zone, namely, link #2.

But hang on; what is the difference between a scope and a zone then? A scope is an abstract amount of topological space, such as *a* link, *a* site, *a* universe. A zone, by contrast, is the specific area where a particular IPv6 address is relevant, such as link #1 in the Nuclear Bike Team, *Yoyodyne*; the *Yoyodyne* corporate network site; our Universe.

> If you are familiar with object-oriented programming basics, you can easily see the following parallel: A scope is to a zone what a class is to an object.

IPv6 address scoping we have just "developed" has the following feature to take note of: *The scope of an IPv6 address can be clearly identified by its binary prefix* (*FE80::/10* is link-local, *FEC0::/10* is site-local, etc) *whereas its zone depends on the network configuration particulars.* A zone ID isn't encoded in the binary value of an IPv6 address: It needs to be stored and conveyed in addition to the address proper, e.g., the way we did it above by extending each link-local address with a specifier, "on link such-and-such".

Does this mean that some form of a unique zone ID is going to be an integral part of every IPv6 address and a field needs to be reserved in the IPv6 header for the source zone and another one for the destination zone? To answer this question, we need to accept and grasp the simple fact about address scoping that *addressing between different zones of the same scope*[4]

[4] Or, simply put, of the same size.

is impossible because, were such zones to be joined up, addresses in their union would no longer be unique. In other words, *a packet whose source or destination address is from a particular zone must never leave that zone* because the address will lose its meaning if that ever happens. What we have just discovered is probably the most fundamental feature of the IPv6 scoped address architecture. Henceforth we will call it "the zone isolation principle".

> In fact, it was obvious to us from the beginning that the user workstations in the *Yoyodyne* corporate network scenario from Fig. 2.2 would be unable to communicate across the LAN boundaries if using link-local addresses only. What we have done now is just generalized this observation to any scope.

Thanks to this feature, an IPv6 node, be it a host or a router, can always know which zone an inbound IPv6 packet came from. For instance, should a packet be received from link #2 with a source address of *FE80::DADA* and a destination address of *FE80::F00D*, the *Yoyodyne* corporate server from Fig. 2.2 will immediately assign those addresses to the zone of link #2 so that they are no longer just bare bits: They will be readily qualified as "*FE80::DADA* on #2" and "*FE80::F00D* on #2" by the server. Consequently, no link indices will be required in the packet header on input and so it will be pointless to include them on output.

Some ambiguity can arise when a node is about to transmit an IPv6 packet. It boils down to choosing the outgoing interface because it is by its interfaces that the node connects to links and, through them, to zones of larger scope. Two quite different cases are possible here:

1. This node is intermediate in the packet's path.
2. This node is the originator of the packet.

In the first case, the node has received the packet through a particular interface and, knowing its ingress interface, qualified its source and destination addresses with their zones as just discussed. The node therefore should be able to pick the right egress interface for the packet. To give an example, a packet with a link-local destination address can only return to the same link it came from—via any interface connected to that link if the node has got multiple connections to it.

It is only in the second case where the destination address zone isn't readily known to the node and a zone ID has to be provided in some form along

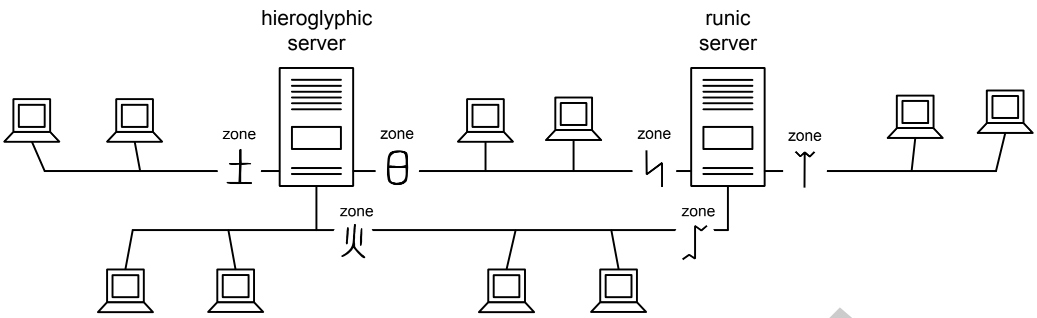


Figure 2.4: Zone names can be as unconventional as runes or hieroglyphs because they have only local significance

with the destination address. For example, should the *Yoyodyne* network admin need to *ping* host *FE80::Baa1* on the Philosophers Stonemasonry Team LAN (link #3) from the server, the target zone will have to be specified in the command line along with the numeric address. This puts a standardized way of zone specification on our to-do list.

We have just considered all of the principal cases where a node needs to determine the zone of an IPv6 address: packet reception, origination, and forwarding. In each case, all information required by the node proved to be available locally, needing no outside information from other nodes. Therefore *any form of zone ID would be redundant in the IPv6 header because plain, 128-bit, source and destination addresses are sufficient there for unambiguous operation of the scoped address architecture.*

> Roughly speaking, the explicit inclusion of zone IDs in the IPv6 header would be a disguised way to give up on the scoped address architecture and simply expand the IPv6 address by a few more bits.

That being the case, zone identification can be safely made private to each IPv6 node, different nodes never having to share their zone IDs with each other. For instance, if there were two servers in the *Yoyodyne* corporate network, then one of them could be referring to directly connected link-local zones by Ancient Chinese hieroglyphs and the other, by Old Germanic runes notwithstanding shared links, as illustrated in Fig. 2.4. It is only required that a node consistently use its own zone IDs or, in a more formal language, that each zone ID remain significant and unique within the node.

> As an off-topic exercise, the reader will locate the exotic symbols from Fig. 2.4 in the current version of Unicode.

> We may seem to contradict to what we said in Section 2.2 in that runic or hieroglyphic zone names are non-ASCII. The point we are now trying to make is that a local zone name will never have to be entered into another node's configuration because over there it would make no sense at all. That's why, at least in theory, a node is free to adopt any zone naming scheme it can handle—just because no other node cares about it.

Although an implementation can choose any suitable way to keep track of connected zones, it makes sense for us to come up with a conceptual method our further discussion can be based on. To be able to reliably distinguish different zones, a node needs to do as little as this:

- remember the scope of each connected zone: link, site, . . . , universe;
- number connected zones of the same scope as #1, #2, and so forth.

The combination of a scope and a zone number within that scope will be known as a **zone index**. Our early references to link-local zones in the above examples as "link #1", "link #2", and so on were essentially a form of zone indices.

Now we have enough concepts and terms under our belt to proceed further. An important decision made above was to not limit the available IPv6 address scopes to link-local only. However, we have so far done very little to support it: The examples given earlier in this section were all based on link-local zones and never mentioned a zone of larger scope. But, as it will turn out, we happened to be lucky in that we didn't rush to consider larger scopes, or else we would probably have told a load of nonsense. The issue is that the introduction of larger scopes entails a few non-trivial questions we are to find answers to first. The basic idea looks simple on the surface: A zone of larger scope can contain one or more zones of smaller scope in it. For example, a site can consist of one or more links. But a rigorous analysis of the matter can't but bring up at least the following two questions:

- Can a zone of smaller scope be shared by two or more zones of larger scope?[5]
- Can the source address and the destination address of the same IPv6 packet come from different zones?

[5] An equivalent way to ask the same question can be this: Can zones overlap?

> Although we haven't discussed the IPv6 header yet, it is already clear that there will be a source address and a destination address in it.

To answer the first of these questions, suppose there is a link, L, shared by two sites, S_1 and S_2. In that case, there can be two nodes, N_1 and N_2, that are connected to link L and have the site-local address, say, *FEC0::C001*. Of course, the zone of N_1's address will be site S_1 while that of N_2's address will be site S_2 and that's how they will be different addresses in spite of the same binary value. Now assume that a third node from site S_1 transmits a packet with a destination address of *FEC0::C001* into link L. According to our working model, that address will not be qualified with a zone index in the packet header. The end result will be that the packet will be received and consumed by both nodes, N_1 and N_2, although it was intended only for node N_1 by the zone isolation principle. In this scenario, the destination address became ambiguous. How can its uniqueness be restored?

That will happen if, and only if, the header and the ingress interface of an incoming IPv6 packet together can unambiguously tell the packet's source and destination address zones [46, Sections 7 and 8]. An IPv6 address from the packet header alone can only tell its scope but not its zone, so the ingress interface information has to fill in the gap and provide for a mapping from the address scope to its zone. This mapping will be univocal and defined for any address scope only on condition that *each IPv6 interface belongs to exactly one zone of each possible scope, regardless of what IPv6 addresses are actually assigned to it* [46, Section 5]. The last example was at conflict with this rule since each of the interfaces connected to link L was effectively in two site-local zones at once.

> However counter-intuitive it can be, an interface can belong to a zone even if it has no IPv6 address from that zone assigned to it. For instance, an interface with just a link-local address on it still belongs to the global zone and can send and receive global traffic, e.g., if the node does packet forwarding.

> Was it a slip to say in the above rule, "exactly one zone", rather than, "not more than one zone"? In fact, no it wasn't; otherwise the mapping of the {IPv6 address, ingress interface} tuple to a zone would no longer be defined for any address, some addresses in received packets mapping to no zone.

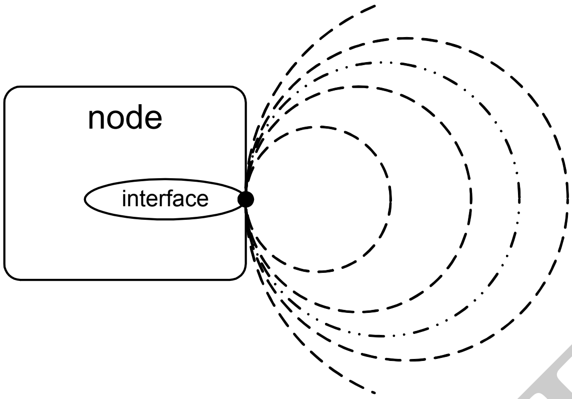


Figure 2.5: IPv6 interface belongs to a zone of each possible scope

> We have to realize that all possible address scopes aren't even known to us. For now we can assume there is a continuum of them as illustrated in Fig. 2.5, where zones of different scope are visualized as a family of tangent circles: A new circle can always be drawn between each pair of existing ones. As a philosopher would put it, a zone exists on the interface regardless of whether it has ever been thought of.

The requirement of one zone per scope per interface we have just come up with can be satisfied only if the network topology of zones has certain properties, namely:

- Zone boundaries cut through nodes, not through links.
- There is no partial overlap between zones.
- A zone of larger scope can fully contain zones of smaller scope.

> A zone can't contain another zone of the same scope, but it is alright for two zones of different scope to have the same boundary. For example, imagine that *Yododyne*'s rival company *Bogodyne* has a corporate network comprising just a single LAN. In that case, the borders of the site-local zone and the link-local zone will coincide in the *Bogodyne* network.

Another interesting point is that these topology rules allow for a smaller zone to belong to two or more larger zones but none of the latter can be of the same scope. For example, a link can belong to a site, a planet, and a universe at the same time, but it can't be shared by two planets. Should two planets need a cross-connect, they will have to establish a border router or, alternatively, a buffer link owned by neither of them.

In addition, the zone isolation principle provides an extra topological constraint:

- A zone needs to be coherent, or "convex", from a routing perspective in that the entire path of a packet whose source or destination address is from a particular zone must stay within that zone boundaries.

The physical map of a network can allow for different paths between a pair of interfaces and some of them may violate the last rule if packets start traveling by them. Therefore it is up to the network admin to exclude such invalid paths from the routing configuration. For instance, site-local traffic should never be routed via a neighboring organization as shown in Fig. 2.6. In turn, the IPv6 stack should be able to detect such zoning violations on per packet basis and block them.

To have a better grasp of all these requirements, examine the complex zone map in Fig. 2.6 and see how it complies with each of the four.

With the zone topology requirements known, we have enough background to approach our second question, which was regarding the relation between the source zone and the destination zone of an IPv6 packet. At first glance, those zones may seem to need to be the same; but in reality the zone isolation principle leads to no requirement so strict. What is actually required, and sufficient, for an IPv6 packet to comply with this basic principle is that its source *interface* be in its destination address zone and its destination *interface* be in its source address zone.

At the same time, it is a fundamental feature of the IPv6 scoped address architecture that the source interface automatically appears in the source address zone and the destination interface, in the destination address zone.

The same idea can be put differently for a better insight into it. Think of an IPv6 packet as an aircraft that can fly only a certain distance known as its

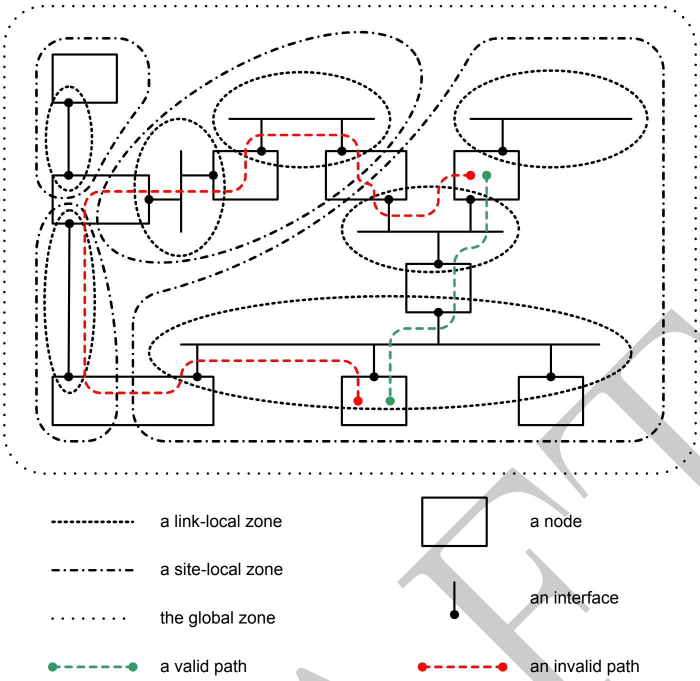


Figure 2.6: A complex zone topology. Valid and invalid site-local paths in it

range. The range of the IPv6 packet is limited by two zones, i.e., those of its source address and destination address, respectively. By the zone topology rules, either the borders of those zones are the same or the smaller zone is fully contained in the larger one. In the latter case, the more significant limit on the packet's range will be imposed by the smaller zone and so the smaller zone will be the limiting factor. Therefore, for the packet to be able to reach its destination, both the source interface and the destination interface need to belong to the smaller of the two zones.

For instance, in Fig. 2.7 two nodes, A and B, are directly connected with a link. In that scenario, node A can send a packet to node B using a link-local source address and a global destination address. Thanks to the zone topology rules, node B won't be confused by such a combination of addresses as long as the ingress interface of the incoming packet is known. Thus node B will be able to respond to A's packet using the same addresses put back to front, i.e., the global source and the link-local destination, even if B has more than one interface. In turn, node A will also be able to link the addresses in B's response packet to their respective zones by following the ingress interface pointer.

To reach its destination interface, an IPv6 packet may need to pass through one or more routers. What will their rules of handling scoped ad-

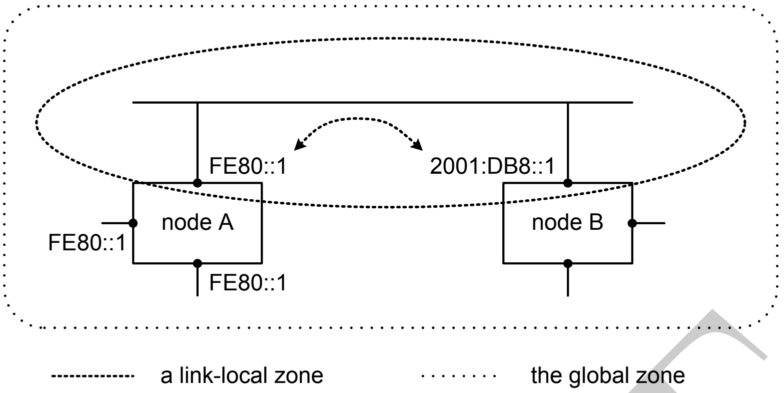


Figure 2.7: Packet exchange using addresses from different zones

dresses be like? An IPv6 router can link the source and destination addresses of an incoming packet to their respective zones using the same rule as outlined above: Either numeric address indicates its own scope and the ingress interface narrows it further down to a particular zone of that scope. Doing it back to front is as possible: The ingress interface links packet's addresses to a family of zones and a numeric address selects one of them by its unique scope.

> Note well that it is the *ingress* interface of an IPv6 packet that links its destination address as well as its source address to a specific zone.

Afterwards the router determines the next hop and the egress interface of the packet using the routing table and, possibly, some additional policy rules. With address scoping in effect, the egress interface can't be any since its choice must satisfy the zone isolation principle. It means that the router must forward the packet in no way that would make it leave its source or destination address zone. And once again the zone topology makes compliance with this rule easy: To keep a packet within its source and destination zones, the egress interface chosen just needs to belong to both of them at the same time.

If there are multiple viable routes for the packet but not all of them comply with the zone isolation principle, it will make sense for the router to limit the working set of routes to the compliant ones only. And if such a working set ends up empty, the packet will have to be discarded due to no valid route.

The reference scope-aware IPv6 router described in [46, Section 9] keeps a separate routing table for each of its connected zones. This method allows the router to immediately rule out the routes that would violate the destination zone of a packet.

According to [46, Section 9], the reference IPv6 router is to handle the forwarding of a transit packet in these steps: First, it is to look up the next hop and the egress interface by the destination address in the zone-specific routing table. Then the router is to verify that the source zone is kept as well and discard the packet if it isn't. Thus the source zone has no effect on which route is ultimately chosen even if the lookup by the destination address returned several viable routes to choose from based on other criteria. Due to this minor shortcoming, it appears possible to construct a proof-of-concept network where the reference router implementation will be discarding packets although it could have forwarded them without zone violation, had it considered the source zone as well when making its next hop decision. The reader will design such a network on paper as an exercise using link-local, site-local, and global scopes.

Now that we have mastered the fundamental structure of IPv6 address scopes and zones, it is time we kept our promise and provided the *Yoyodyne* network admin with a means to selectively *ping* host[6] *FE80::Baa1* on the Philosophers Stonemasonry Team LAN, which is link #3 in Fig. 2.2, from the corporate server. What information is missing from this command line: `ping6 fe80::baa1`? Is it the zone index that is misplaced?

As we will find out in Section 4.3, IPv6 has a control protocol different from IPv4's ICMP. Due to this, what was *ping* in IPv4 is known as `ping6` in IPv6. That's the practice; but in theory introducing a new utility wasn't strictly necessary: The same `ping` utility could have been extended to detect the target address version and select the appropriate protocol from that. For instance, the *JunOS* CLI went the latter way; but internally its `ping` command still invokes */sbin/ping* or */sbin/ping6* depending on the target address version.

As we concluded above, a numeric IPv6 address has its scope already encoded in it, and in this example it is as clear as day that the address has

[6] As we are to learn in Section 2.6, he will be pinging a particular interface in that host because an IPv6 unicast address belongs to a specific interface rather than to the entire host. This was de facto true in IPv4, too, but hardly supported by the standards.

link-local scope because it has a prefix of *FE80::/10*. Therefore a complete zone index such as "link #3" would be redundant here. To save the admin extra typing, let's review our discourse and solve the following riddle: What will remain if the numeric address is taken out from a full zoned IPv6 address? Apparently, it will be a network interface in the address zone. Indeed, it is keeping track of the ingress interface that enables an IPv6 node to link an address in an incoming packet to a specific zone. Then the most natural candidate for an ID string to extend a numeric IPv6 address to a full zoned one will be the system name of a suitable network interface connected to the target zone. For example, assuming link #3 is connected to the *Yoyodyne* server via interface `en2`, the fully qualified address "*FE80::Baa1* on link #3" can be encoded as follows:

```
ping6 FE80::BAA1%en2
```

In this notation, the numeric address is separated from the interface name with a percent sign, a symbol otherwise rarely encountered in the network ABC.

> Admittedly, this choice of the separator character isn't perfect. For one thing, the percent sign is a metacharacter in the URI format and so a zoned IPv6 address can't be inserted into a URI literally unless extra provisions are made. An optimal solution for this issue still is in the works [47, 48].

To use interface names for zone identification sounds like an ingenious trick, but before we go ahead and make it standard, let's place it under scrutiny, looking out for inconsistencies. The most obvious gap is this: In order to work out the zone topology rules, we considered inbound packets and ingress interfaces; but now we have leaped to an outbound packet and its egress interface. Was it a valid step? For an inbound packet, all of the mappings we relied upon were unambiguous: An inbound packet could only be received through one interface[7] belonging, in turn, to just one zone of each scope. However, those relations weren't designed to be one-to-one and so they can't just be reversed and applied to packet origination. In actuality, several of the local node's interfaces can belong to the same zone. For example, the node can be connected to the same link with more than one interface for fault tolerance or load balancing and, furthermore, all of the node's interfaces can end up in the same site.

Therefore an accurate interface-to-zone map can be charted for a network only by its designer. At the same time, using system interface names to

[7] Perhaps in the quantum networks of the future this will no longer hold true.

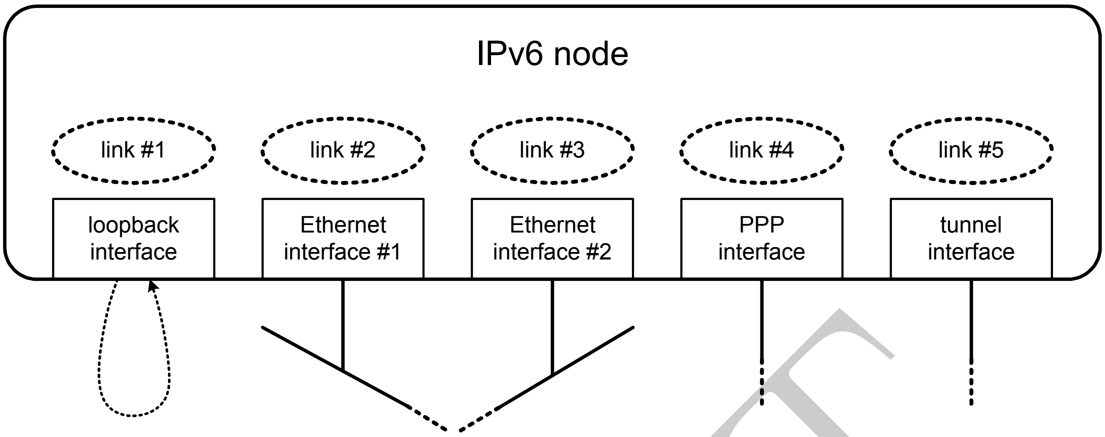


Figure 2.8: Initial node configuration assuming a one-to-one correspondence between interfaces and zones

identify zones can only provide a reasonable initial approximation to that map, e.g., as shown in Fig. 2.8. Its value is in that the node's IPv6 stack can automatically build it and the network admin will just have to correct it using the actual network zone map [46, Section 6], e.g., as shown in Fig. 2.9.

> Consequently, an IPv6 stack implementation has to provide tools for adjusting the node's view of the interface-to-zone map.

To reflect this feature in the IPv6 glossary, let's refer to the text ID of a zone, to be placed after the percent sign, as a **zone_id** [46, Section 11.2]. Quite often a zone_id will come from a suitable interface name because it is a clear and convenient tag for a zone, but in fact there is no strict rule about that. In the end, the text notation of a fully qualified scoped IPv6 address will look as follows:

```
<numeric_address>%<zone_id>
```

And if a scoped prefix needs to be written as text, its length is to be placed after the address proper, as usual [46, Section 11.7]:

```
<numeric_address>%<zone_id>/<prefix_length>
```

For instance, a link-local address can be entered along with the subnet prefix length as in the following example:

```
FE80::DADA:C001%en5/64
```

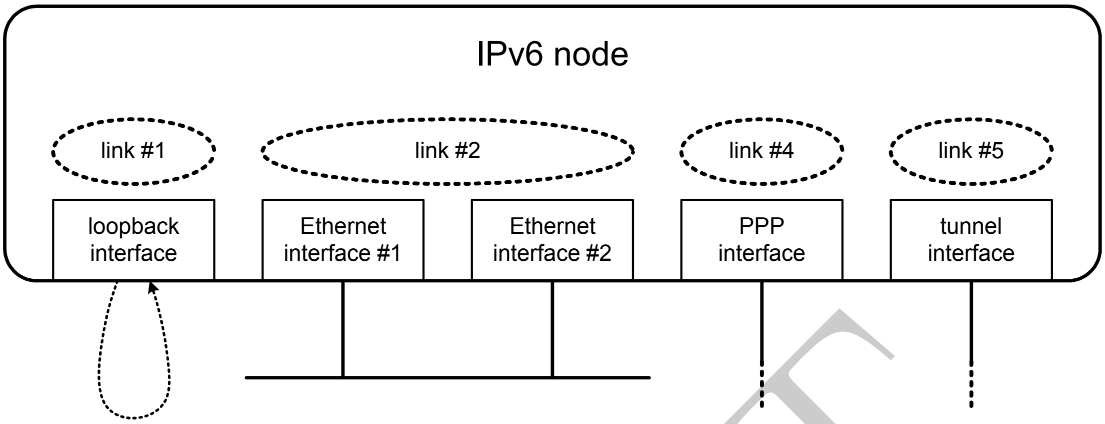


Figure 2.9: Resolving the zone ambiguity of two interfaces connected to the same link

Table 2.2: How scoped *ping* looks like in different OS types

OS flavor	Command
FreeBSD	`ping6 FE80::Baa1%fxp2`
Linux	`ping6 FE80::Baa1%eth2`
MacOS X	`ping6 FE80::Baa1%en2`
MS Windows	`ping6 FE80::Baa1%4`

Since different operating systems stick to different interface naming schemes, it is no surprise that their zone_id styles don't look the same either. For example, the above *ping6* of a *Yoyodyne* host by its link-local address would have to be entered in different OS types about as shown in Table 2.2. This once again illustrates how zone naming is private to each IPv6 node.

To conclude our work on the IPv6 scoped address architecture, we need to answer this practical question: Will an IPv6 address always have to be extended with a zone_id in the node settings as well as in the command line? As it turns out, the answer is: No, it won't. Firstly, there are two special ones among all possible IPv6 scopes:

- The loopback address *::1* has scope of a single loopback interface. A regular IPv6 node will hardly need more than one of those, and so a zone_id can be safely omitted from the loopback address. It can be regarded as a special case of a link-local address that can exist only on the imaginary link attached to the loopback interface [46, Section 6].
- A global address, such as *2001:DB8::1*, has been granted with not only

global but cosmic scope thanks to the Space Age. On that account it needs no zone specifiers—at least until aliens from a parallel universe want to interconnect their IPv6 Internet with ours.

> A comment will be apropos now about the term "scoped address", which can be encountered herein, in the RFC documents, and elsewhere. Please bear in mind that it is not to mean a non-global address. The very essence of the IPv6 scoped address architecture is that *each* IPv6 address except the unspecified one has got a certain scope, and it is just that some addresses happen to have global scope. Therefore a "scoped address" is that whose scope can be any, i.e., an IPv6 address at large. This term is hardly anything more than a reminder to the protocol and implementation developers who come from IPv4 background that they always need to allow for IPv6 addresses having non-global scope and give a careful consideration to each aspect of their work arising from this unusual feature of IPv6 addressing.

Besides, it can be handy to be able to leave zone_id out if its information is redundant. For example, it makes no sense to specify it for a site-local address if all interfaces of the local node belong to the same site; or for a link-local address if the local node has just one interface besides the loopback. To generalize, IPv6 implementations can be recommended to support default zones so that there is a default zone for each practical scope [46, Section 6]. If they do so, the IPv6 stack will be able to guess the zone of an address in case there is no zone_id in it.

> The converse is also true: In certain practical scenarios, it can make sense to attach a specific zone_id to an address having no scope, that is, to an unspecified address of *::*. As we said in Section 2.3, an unspecified address passed down the Sockets API usually indicates a request for the network stack to pick an optimal address for whatever purpose it is supplied for, e.g., for the local endpoint of a TCP connection. These semantics can be extended as follows: If a particular zone_id is specified along, the network stack shall limit its address selection to that zone only [46, Sections 4 and 11.1].

Although we have discussed only unicast address scopes and zones so far, they can be introduced in the same manner for IPv6 multicast addresses as well. With scoped multicast addresses available, a well-known one can be assigned, for example, to the "all IPv6 routers on this link" multicast group. However, a detailed study of multicast address scopes will have to wait until Section 2.8.

2.5 Well-known Applications of IPv6 Link-local Addresses

What we have just “built” is a powerful mechanism for IPv6 address scoping. Will there be more important and widespread uses of it than that to create nested private networks? Scopes larger than a link but smaller than a universe are governed by individual network admins and it is up to them to utilize the scoping mechanism for a particular purpose, thus leaving only link-local scope available for well-known applications. But, as we will see in a moment, that is quite enough.

> Let’s recap what the term “well-known” stands for in the context of TCP/IP. Throughout its history, the Internet has had no standards as such because there has been no body to enforce them—not that one has ever been necessary. Simply put, there is no such thing as a TCP/IP compliance sticker. People are free to adhere to the RFC specifications or take their own way; but if they choose to make up specs of their own, they shouldn’t be too surprised that their system can interoperate with no one else’s. Hence the *well-known* wire data formats, parameter values, peer behavior rules, et cetera: They are merely known to, and voluntarily adopted by, those who want to build a system capable of communication with the world. To us, “well-known” and “standard” are synonyms, but the former is in greater accord with the philosophy of the ’Net.

A benefit of link-local scope is that it is clearly defined, always spanning just one link. Furthermore, every link existing out there is a zone of that scope because IPv6 zones exist regardless of addresses actually assigned to their interfaces. What basic aspects of IP network operation are limited to a link and never cross its boundaries?

First and foremost, it is IP packet forwarding. According to the fundamental IP operation model—and IPv6 is no exception from it—packets travel between nodes via links. Therefore a basic step in a packet’s path from its source to its destination is a hop, that is, the transmission of the packet to another IP node within a link’s reach from the current node. The IP address of the receiving interface is referred to as the next hop address, thus implying that there can be more hops beyond it in the packet’s path.

> These basic facts of TCP/IP were recapped here only to ensure we are on the same page with the reader.

Obviously, it is *sufficient*, but not necessary, for a next hop address to be link-local since the interface selected by it is known to be directly connected to the current node with a link. Of course, there is no problem with using an address of larger scope for a next hop, but the information contained in such an address will be redundant for this purpose.

If being a router can be regarded as one of an IP node's possible roles, it will consist in transit packet forwarding whereas packet origination and consumption are handled by the host role. By what we just said, the router role only requires link-local addresses while those of other scope aren't essential for it. The beauty of using link-local addresses for packet forwarding is in that the addresses assigned to the network's forwarding plane[8] get decoupled from the addresses used for end-to-end communication by hosts. For example, should a company change its ISP and receive a different IPv6 address block, it will no longer have to renumber all of its router interfaces. To encourage this practice, we will establish the following requirement: *Each interface of an IPv6 router must have at least one link-local address on it* [46, Section 2.3].

Yet another possible use for link-local addresses is that for address autoconfiguration purposes in hosts. Indeed, using link-local addresses can be an effective solution to the chicken-and-egg problem that a host can't receive its address configuration from the network until it has assigned a valid address to its interface. The autoconfiguration framework deserves an extended discussion so we will revisit it later on, in Section 5.4, and for now we will only conclude that *an IPv6 host will need address autoconfiguration support and the latter requires that a link-local address be assigned to each of the host's interfaces* [50].

To sum the above up, *each IPv6 interface must have at least one link-local unicast address assigned to it* [21, Section 2.1]. In addition, the interface will be likely to need unicast addresses of other scope as well, and this can be generalized to just having multiple unicast addresses assigned to the same interface—a feature that has been around since the IPv4 times [51, Section 3.3.4.1]. To make this kind of configuration possible with no restrictions, *a single IPv6 interface can have multiple unicast addresses of any scope*[9] *or subnet assigned to it* [21, Section 2.1].

[8] Back in the early days, it was called "the subnet", which meant the network less its end hosts [49].

[9] We said "scope" instead of "zone" because two addresses on the same interface can never end up in different zones of the same scope. Conversely, should their scope be the same, they will inevitably be in the same zone. This is by the IPv6 zone topology rules we worked out in Section 2.4.

Simply put, it is OK for an IPv6 interface to have many unicast addresses assigned to it and for some, or all, of those addresses to have the same scope or even the same subnet prefix. A similar conclusion about multicast addresses will follow from this one when we get to what is known as Neighbor Discovery in Section 5.1 because then an IPv6 interface will need a multicast address for up to each of its unicast addresses.

There are far-reaching consequences for other protocols from the decision to support multiple addresses on each IPv6 interface. For example, OSPF for IPv6 (OSPFv3) runs per link [52, Section 2.1], unlike its predecessor OSPFv2 which ran per IPv4 subnet. With multiple subnets present on each IPv6 link, the old way just wouldn't scale.

2.6 IPv6 Unicast Address Structure

In IPv4, the structure of a unicast address was defined by the hierarchical prefix distribution procedure whereby each participant would use a prefix it had been given by the upstream authority to append a few more bits to it from the right and hand the resulting extended prefix one tier down, to a subordinate entity. Ultimately, at the bottom tier, the final prefix would be assigned to a link and become a subnet prefix. Thus, from a bottom-tier participant's viewpoint, an IPv4 unicast address consisted of as many as three bitfields: a prefix assigned from the upstream, a subnet number and a host number within the subnet; the first two fields together would make up the subnet prefix. This format applied as well to non-global IPv4 unicast addresses; e.g., the well-known IPv4 private-use prefixes [79] were also received from an upstream authority, which was *IANA* itself in that case instead of an LIR or an ISP. Lately, the lengths of those IPv4 address bitfields were flexible and not nailed down in the protocol thanks to the advent of CIDR [33]. The assigned prefix length depended on the path down the hierarchy that had led to a specific address. Then the prefix owner would split up the remaining address bits to his or her liking in an attempt to balance fine network partitioning and address conservation, as two valuable IPv4 addresses would be wasted in each subnet.

That scheme had two benefits still of great value to us. First, it allowed to distribute an address space using just one trivial operation, that is, address block bisection. It isn't hard to see that appending a new bit to a given prefix and setting that bit first to 0 and then to 1 was equivalent to splitting the address block represented by the original prefix in two halves of

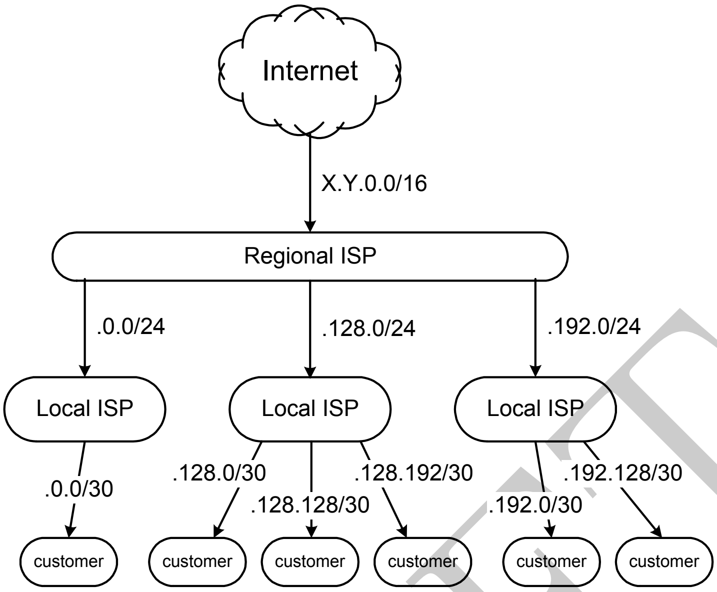


Figure 2.10: IPv4 route aggregation in the Internet

equal size, each represented by one of the two new prefixes. Of course, this procedure had the obvious limitation that it was only applicable to blocks of 2^n IPv4 addresses having a common prefix of $32 - n$ bits; the starting block was the entire IPv4 address space represented by an empty prefix, which corresponded to $n = 32$. Had an address block been just an arbitrary address range, it couldn't have been reduced to a single prefix; hence the limitation. Only thanks to it could IPv4 prefixes and address blocks be regarded as equivalent and interchangeable entities.

Second, superimposing an address distribution hierarchy with a routing hierarchy provided for route aggregation. For an illustration, consider the following scenario shown in Fig. 2.10: A regional ISP secured the IPv4 address block *X.Y.0.0/16* and is now subdividing it into smaller blocks of *X.Y.Z.0/24* and handing those down to local ISPs that use the regional ISP as their upstream carrier. The smaller ISPs, in their turn, provide each customer with a */30*. The end result is that only local ISPs have to care about */30* routes to their customers' networks. For the regional ISP, it is sufficient to install *X.Y.Z.0/24* routes, each of them pointing to an individual local ISP regardless of the number of its customers. And at the topmost level, in the default-free zone of the Internet, the regional ISP can be represented by a single *X.Y.0.0/16* route. In a perfect world, this can be a way to prevent the exponential growth of the routing tables as we go up the routing hierarchy.

As for the apparent drawbacks of the IPv4 address distribution scheme, they had largely to do with a small address space size and the shortage of IPv4 addresses that ensued. On the one hand, a chunk of 2^n addresses often seemed too coarse and improperly sized for a particular purpose when initially allocated, e.g., to a subnet. On the other hand, there was no optimal split of the available bits into the subnet number and the host number fields that would suit a live and growing network because its subnets would fill up notwithstanding all the careful planning and by then it would be too late to extend the subnet with the adjacent block because it was already taken.

> Each IP address block has one and only one adjacent block. This is an immediate corollary from distributing the IP address space by block bisection, equally applicable to IPv4 and IPv6. Working out the adjacent block prefix from a given prefix is as easy as inverting the least significant bit of the prefix given. For instance, prefix `110` is adjacent to `111`, and `1001` to `1000`. It is easy to see that adjacency is mutual: If block A is adjacent to block B, block B is adjacent to block A. Adjacent blocks can be recombined to make up a single block twice as large by just discarding the least significant bit in either of their prefixes, for those prefixes differ by the least significant bit only. By contrast, non-adjacent blocks can never be merged into one address block under a single prefix.

As we move on to IPv6, the drawbacks of the binary address distribution scheme can be alleviated by a much larger address space. At the same time, its merits remain attractive enough for us to want to reuse it in IPv6. Therefore the IPv6 unicast address, too, is going to consist of fields equivalent to the assigned prefix, the subnet number and the host number in its IPv4 counterpart.

Yet again we will start working back to front—in this case, from the address tail—and first consider what used to be the host number field. The modern practice pioneered as early as in IPv4 has been that both routers and hosts can have multiple network interfaces including those in the same subnet [51, Section 3.3.4]. E.g., this can provide a host with a certain level of redundancy in its network connection. For such a configuration to be valid, each of the host's interfaces has to have an IP address that is unique within its zone. In case the interfaces end up in the same subnet, their addresses can only differ in their least significant bits because they are bound to have the same subnet prefix. Therefore they will require different host numbers albeit they belong to the same host. This is a mere confusion of terms we can sort out in IPv6 by renaming the field that was known as "host number" in IPv4 to the more appropriate **interface ID**[10] [21, Section 2.5].

[10] Abbreviated to **IID** in some RFC documents.

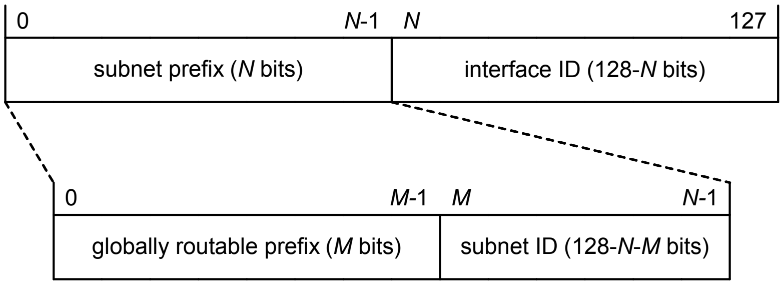


Figure 2.11: Generalized structure of an IPv6 global unicast address

It is emphasized by this name that *an IPv6 address always belongs to an individual interface rather than to an entire host.*[11] For consistency, what was known as "subnet number" in IPv4 will be referred to as **subnet ID** in IPv6. Finally, the most significant bits of an IPv6 unicast address are still occupied by a prefix assigned from the upstream, e.g., a globally routable one as shown in Fig. 2.11. Together the assigned prefix and the subnet ID constitute a **subnet prefix**.

> As for the netmask, it is history. The IPv6 architecture provides no support for non-contiguous netmasks with alternating runs of ones and zeros in them, thus ruling out the case in which the terse prefix length was unable to convey the same information as the long and redundant bit mask. Thanks to that, the largely obsolete netmask concept can finally be ditched altogether.

Now we should recall that we aimed in Section 2.1 for a 64-bit IPv6 interface ID so that it could accommodate an EUI-64. In that case the subnet prefix length would also be 64 bits. However, that was a mere speculation to base our address length choice on, not a strict specification of the protocol. So a seasoned network admin still has the option to pick an interface ID length the way he did it in IPv4, that is, based on his scientific or intuitive expectations of the subnet growth. What can be wrong with this well-established approach? Its problem is that it has proven too easy to err on the side of frugality and allocate too few bits to the interface ID, particularly if burdened with IPv4 address space conservation experience. To prevent IPv6 subnet overpopulation once and for all, the human factor needs to be eliminated here. For this sake, the IPv6 interface ID and subnet ID lengths will both be nailed down at a fixed value of 64 bits [21, Section 2.5.1].

[11] This has been de facto true in IPv4 as well but, as far as we know, never made it into a protocol update such as a Host Requirements RFC.

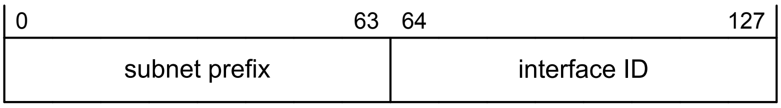


Figure 2.12: Actual format of an IPv6 unicast address

The IPv6 address space is large enough for this decision to be affordable. In return, the network admins will never again have to waste their time on working out unreliable forecasts of subnet population growth: As long as EUI-64 lasts, they will have enough IDs for new interfaces. And what about a point-to-point link connecting just two interfaces by definition? If anyone maintains that a */64* can be a bit too much for it, here is a way to counter their argument: "And what if tomorrow you have to add a third interface to that point-to-point link and thus convert it to a broadcast one? Will you enjoy the hassle of renumbering it?"

A typical case of extending a point-to-point link is encountered when an inline security appliance needs to be inserted into it.

This way the interface ID length is decoupled from the link-layer technology as well as from the projected subnet size. Later on we will see how having the interface ID length fixed opens up other interesting opportunities for protocol design.

Today the rule that an IPv6 interface ID be 64 bits long has ended up having an ambiguous status.

Originally, it was a pure policy decision. That is, no technical detail of IPv6 was to depend on it. For all technical purposes, the interface ID length was variable, say, N, which could be set individually for each subnet. And it was only in the final step that N was fixed at a value of 64 for *all* IPv6 subnets. (Imagine a machine with a control lever fixed in a specific position although, by its design, the machine would have no problem operating at *any* lever position.) However, certain features were later introduced in IPv6 that could operate only as long as $N = 64$. (It is as though an extra part were added to our imaginary machine that would cause a crash as soon as the lever were moved out of its fixed position.) Among those features are Unicast Based Multicast addresses (Section 2.9), Cryptographically Generated Addresses (Section 6.3) and Hash-Based Addresses (Section 6.8)—we are going to meet all of them in due course.

On the other hand, everyday practice calls for using longer prefixes on point-to-point links known to never be extended. A useful prefix can be up to */127*, which corresponds to $N = 1$. This point of view also has proponents whose arguments are quite strong [53].

So the matter of when and how strictly the IPv6 interface ID length is to be governed by the protocol essentially remains a research topic. This is normal for a live and developing protocol. Meanwhile, for the purpose of our discourse we can safely assume $N = 64$ and this won't be at odds with the current IPv6 practice.

A minor reservation is to be made that, among all IPv6 unicast addresses, there is a single prefix reserved for special purposes, so its addresses don't need to have a 64-bit interface ID. The bitwise value of that prefix is `000` or, in the IPv6 notation, *::/3* [21, Section 2.5.4].

As for link-local addresses, they will be no exception from the general rule of a 64-bit interface ID. At the same time, their well-known prefix *FE80::/10* is 10 bits long, thus leaving 54 bits to the subnet ID. What will its value be? Let it be just zero. Other values of that field are reserved in link-local addresses [21, Section 2.5.6]. Thus link-local addresses are effectively limited to the *FE80::/64* subnet as shown in Fig. 2.13.

2.7 IPv6 Interface IDs and EUI-64

When introducing a fixed-length, 64-bit, interface ID in Section 2.1 and Section 2.6, we had it in mind that an EUI-64 link-layer address should be

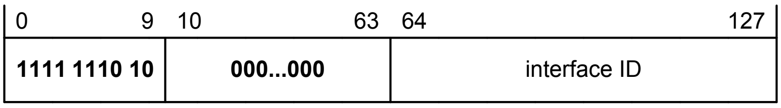


Figure 2.13: IPv6 link-local address format

Table 2.3: Rules for mapping EUI-48 and MAC-48 to EUI-64

Address type	Original address pattern	EUI-64 pattern
EUI-48	*xx-yy-zz-ww-vv-tt*	*xx-yy-zz*-`FF-FE`-*ww-vv-tt*
MAC-48	*xx-yy-zz-ww-vv-tt*	*xx-yy-zz*-`FF-FF`-*ww-vv-tt*

directly embeddable in it. Such embedding would be a great feature to have because an EUI-64, if it was received from *IEEE*, is supposed to be globally unique. For example, a rough procedure for an IPv6 node to auto-assign a unique IPv6 link-local address[12] to its interface could boil down to the bit-wise concatenation of the IPv6 link-local subnet prefix *FE80::/64* and the interface's own EUI-64 address. For instance, the interface whose EUI-64 were *00-11-22-33-44-55-66-77* would get, to a first approximation, the IPv6 link-local address *FE80::11:2233:4455:6677*.

However, not all link-layer addresses are EUI-64 yet. How are other link-layer address types to be handled? They will pose no problem as long as their mapping to EUI-64 can be defined. Provided that the original link-layer address has less than 64 bits in it, that mapping can be injective; that is, different original address values can be mapped to different EUI-64 values because there is enough room for that in the EUI-64 address space. In network protocol design, the clearest and handiest family of such mappings will be an encapsulation of the original link-layer address into EUI-64, whereby the original address bits are just put at specific locations in the resulting EUI-64.

For example, the following simple mappings have already been defined for EUI-48 and MAC-48 [23]—see Table 2.3. In those mappings, two extra bytes of a specific value appear between the OUI and the extension identifier: `FF-FE` in EUI-48 or `FF-FF` in MAC-48. E.g., the EUI-48 *02-35-AB-00-64-C1* will be mapped to the EUI-64 *02-35-AB-FF-FE-00-64-C1*—the extra bytes inserted are underlined.

> Hence network equipment vendors are to issue no EUI-64 extension identifiers starting with `FF-FE` or `FF-FF`.

[12] Of course, it would only need to be unique within the link, not globally.

Hang on a minute—what is the difference between EUI-48 and MAC-48 anyway? Before we have had a chance to confuse this matter up, let's see what its modern interpretation actually is because we will need it by the end of this section. Both address types are managed by *IEEE*. Originally, MAC-48 were to be the IEEE 802 addresses, and EUI-48 were to be used as identifiers in other technologies. At first, *IEEE* even assumed that those address spaces could be different notwithstanding the same format.

> As a cynic would comment, *IEEE* seemed to be going to sell the same bytes twice.

Nevertheless, the current document posted by *IEEE* clearly states that MAC-48 is an obsolete designation for what is just a subset of EUI-48 [54]. Therefore from now on we can be saying, "EUI-48", to imply, "MAC-48 and whatnot". Moreover, the mapping from MAC-48 to EUI-64 can be disregarded in favor of the EUI-48 to EUI-64 rule.

> A scrupulous reader can argue that an address and an identifier are not the same thing. Granted, it is generally true, but an identifier can still be used as an address if certain conditions are met. For example, in the classic Ethernet technology a station ID could serve as its address owing to the fact that *Ethernet* was a broadcast media. Thus an Ethernet address didn't need to be a locator providing information on the station's whereabouts in the LAN. This feature couldn't but affect the subsequent development of switched *Ethernet* and resulted in MAC address auto-learning being essential for the latter.

Now that we know what EUI-48 is and how it is related to MAC-48 and mapped to EUI-64, let's get back to our current matter and consider the properties of a hypothetical IPv6 address that would result from the concatenation of a */64* subnet prefix and an EUI-64. By the uniqueness of the EUI-64, we would like to presume the resulting IPv6 address unique too; but can its uniqueness be really guaranteed when another node can have the same address assigned manually?

Here is an example to illustrate our concern. Suppose that there are two IPv6 nodes, A and B, connected to the same link with their respective interfaces. Clearly, either interface will need a link-local address unique within the link. Suppose that node A's link-local address is assigned by us while node B's is auto-assigned by the node itself based on the EUI-64 of its interface. Now we have allocated the address *FE80::11:2233:4455:6677* to node

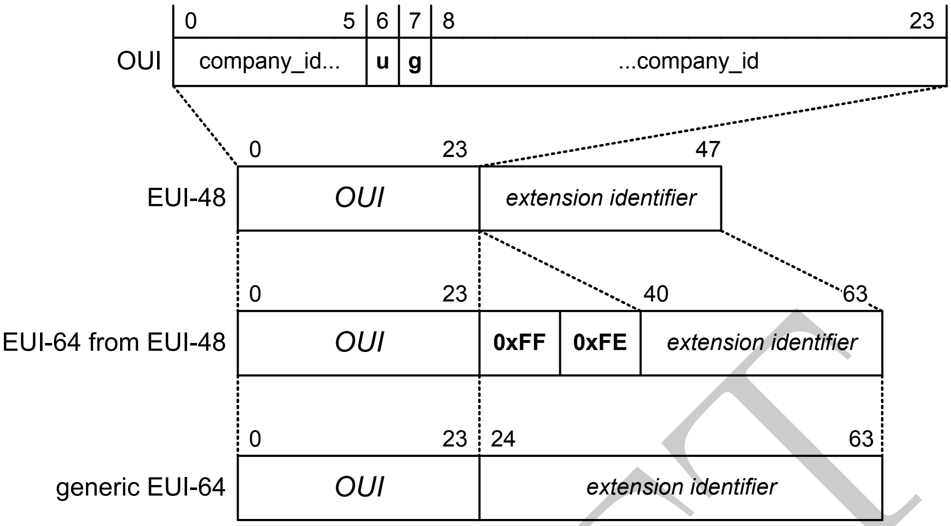


Figure 2.14: OUI format and mapping between EUI types

A. How can we ensure that node *B* won't pick the same address? Shall we check beforehand that its EUI-64 isn't *00-11-22-33-44-55-66-77*? Obviously, such "auto-assignment" is no good at all because we have to verify its proper functioning by hand. If we had a thousand nodes rather than just two of them and 999 of them were auto-assigning their addresses to themselves, we would have to check each of the 999 auto-assigning nodes to be able to manually assign a unique address to the thousandth one. That would be a painstaking job! Can it be avoided with a clever address design?

First of all, let's recall what we knew about an EUI-48 address, or MAC-48, if you will. It had a certain structure and, fortunately, that structure has been transferred with minimal changes to EUI-64; only the extension identifier field has grown longer by two bytes—see Fig. 2.14.

At the moment, of particular importance to us in the EUI-64 format is the **U/L** bit, aka **u**, which is there to distinguish **u**niversally administered addresses with an OUI assigned by *IEEE* from those assigned **l**ocally by the network admin or whoever. The other bit flag there, which we will consider later, is **I/G**, aka **g**, to indicate whether the address is **i**ndividual or **g**roup. The EUI-48 to EUI-64 mapping preserves both bits as they are.

Thanks to the **U/L** bit, we can at least avoid conflicts with EUI addresses assigned by the network hardware vendors. E.g., the EUI-64 *00-11-22-33-44-55-66-77* has that bit cleared (0), so it is universally administered by *IEEE* and we had better steer clear of it when coming up with an interface ID for manual assignment. By contrast, the EUI-64 *02-11-22-33-44-55-66-77* has the **U/L** bit set to 1, so it is locally administered and hence safe to use as a manually assigned interface ID since it will never appear on some network interface as its stock link-layer address.

To sum the idea up, an IPv6 interface ID needs to be based on EUI-64 and inherit its format elements. However, if we apply the EUI-64 format verbatim, that will rule out manual assignment of short and handy interface IDs such as 1, 2, 3—as in *2001:DB8::1*. To see why it is so, consider for instance the EUI-64 with a numeric value of 1, that is, *00-00-00-00-00-00-00-01*. Its **U/L** bit is set to 0 indicating a global address unsuitable for local assignment. For this reason, we would always have to set the **U/L** bit in interface IDs assigned by hand; e.g., the clumsy *2001:DB8:0:0:200:0:0:1* would have to be substituted for the compact and easy-to-remember *2001:DB8::1*.

There still is a way to avoid this inconvenience. What can save the day is a one-to-one mapping between the genuine EUI-64 and the IPv6 interface ID. Indeed, we aren't obliged to stick to the EUI-64 format: All we are to do here is preserve the information contained in the EUI-64, and a one-to-one mapping will do the job. The exact mapping we are after is as trivial as inverting the **U/L** bit so that nice interface IDs such as 1, 2, 3 become available for local administration. The resulting format is known in IPv6 as **Modified EUI-64**. Its only difference from the *IEEE* EUI-64 is that in the reversed meaning of the **U/L** bit values: Now 0 means "local" and 1 means "global / universal" [21, Section 2.5.1 and Appendix A].

So the final form of the requirement we are after will be this: *An IPv6 interface ID must be constructed in Modified EUI-64 format.* This rule applies to all IPv6 unicast addresses except those from the reserved *::/3* block [21, Section 2.5.1].

In order to make an IPv6 address from an EUI-64 by this rule, the **U/L** bit needs to be inverted and the appropriate 64-bit subnet prefix is to be prepended to the resulting Modified EUI-64 so that it appears in the high-order bits of the IPv6 address constructed, where it belongs.

The respective recipe for EUI-48 (MAC-48) can be just a combination of the two steps already discussed: first from EUI-48 to EUI-64, then from EUI-64 to IPv6, as follows:

1. EUI-48 to EUI-64:

   (a) Insert the two-byte sequence `FF-FE` between the 3rd and 4th bytes of the EUI-48 address, that is, between the OUI and the extension identifier.

2. EUI-64 to IPv6:

   (a) Invert the **U/L** bit.

(b) Prepend the subnet prefix.

Those who enjoy exploring the history of network protocol design will find it interesting that MAC-48 and EUI-48 were regarded to be different entities when the Modified EUI-64 format and the MAC-48 conversion rule were introduced for the first time. Back then the IPv6 pioneers misread the *IEEE* documentation and confused the EUI-64 encapsulation rules for EUI-48 and MAC-48. They intended to be speaking of MAC-48, which was natural to IEEE 802 LANs including *Ethernet*, but prescribed using the EUI-48 rule instead and it was the latter than ended up in the IPv6 stack implementations—see the note at the end of [21, Appendix A]. Later, there was some discussion whether that error should be corrected [55, 56], but it was deemed less disruptive to leave it as it was. And now that EUI-48 and MAC-48 have finally become one, the old error has sorted itself out and turned into a smart choice.

Here are a few examples of mapping EUI addresses to IPv6 ones:

1. The EUI-64 *00-11-22-33-44-55-66-77* and the subnet prefix *2001:DB8::/64* make the IPv6 address *2001:DB8::211:2233:4455:6677*.

2. The MAC-48 *10-20-30-40-50-60* and the link-local subnet prefix (*FE80::/64*) make the IPv6 address *FE80::1220:30FF:FE40:5060*.

Reverse analysis of IPv6 unicast addresses is as possible:

1. The IPv6 address *2001:DB8::D00F* was manually assigned or based on a locally administered EUI-64 because the **U/L** bit is unset (zero) in its interface ID.

2. The IPv6 address *2001:DB8::200:FF:FE00:D00F* was constructed from the global EUI-48 or MAC-48 *00-00-00-00-D0-0F*.

3. The IPv6 address *2001:DB8::200:0:0:D00F* was constructed from the global EUI-64 *00-00-00-00-00-00-D0-0F*.

Given an IPv6 unicast address, will you be able to determine if its interface ID complies with the Modified EUI-64 format or violates it? (Hint: *It doesn't seem possible. One has to know how the interface ID was created to be able to tell whether it was the right or wrong way.*)

Although nearly all local interface ID values, i.e., those with **U/L** of zero, are available to network admins and users, there still are several IDs among them that are reserved for special use [80]. Later on, we will be able to see precisely why they have to stay reserved. In particular, the zero interface ID, `0000:0000:0000:0000`, is reserved.

Having started this section with a case for IPv6 address autoconfiguration, we can't conclude it until the following question is answered: Are the rules just worked out quite *sufficient* for an auto-assigned IPv6 address to be unique? Consider the following scenario. There are two interfaces connected to the same link, denoted by A and B. Interface A has a *locally administered* EUI-64 on it, say, *02-00-00-00-00-00-00-01*, which is perfectly legal. The network admin goes on to assign the IPv6 link-local address *FE80::1* to the other interface, B, which breaks no rule either because that address complies with the requirement for an IPv6 interface ID to be a Modified EUI-64. Finally, the node owning interface A auto-assigns a link-local address to that interface based on its EUI-64. It is easy to see that auto-assigned will be the same address *FE80::1*, resulting in an IPv6 address conflict in the link-local zone.

In depth, this problem is due to locally assigned IPv6 interface IDs implicitly taking slots in, and so encroaching, the locally administered EUI-64 address subspace.

Here we have at least one problematic case linked with using locally administered EUI-64. Therefore IPv6 address auto-assignment will remain unreliable and prone to conflicts until we focus in due time on a specific mechanism for address clash prevention. That said, the introduction of IPv6 interface IDs based on the Modified EUI-64 format lays the necessary groundwork for more advanced features of IPv6, as we will witness in Section 5.4.

2.8 IPv6 Multicast Address Structure

In Section 2.3, we touched on IPv6 multicast addresses and determined that they will exclusively occupy the prefix *FF00::/8* and have scopes similarly to their unicast counterparts. Now it is time we settled the rest of their technical details.

To begin with, let's realize that there are going to be two large categories of multicast addresses: well-known ones and those available for private purposes of the community. The former addresses are to be assigned by *IANA* from the corresponding registry [57]. Such assignments are long-term if not

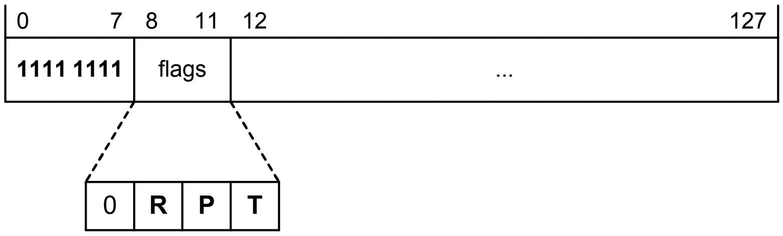


Figure 2.15: IPv6 multicast address flags field

forever, so the addresses in this category can be referred to as **permanently-assigned**. To bridge it with our IPv4 experience, the addresses *224.0.0.1* (All Systems on this Subnet) and *224.0.0.2* (All Routers on this Subnet) effectively fell in this category [58]. Contrastingly, the addresses from the other category can be borrowed by anybody for their own purposes and freed up later when they are no longer needed. Such addresses will be known as **transient**. Drawing a clear distinction between the two categories is important because well-known addresses will be used by standard protocols and it will be too easy for an address conflict across the two categories to disrupt network operation.

We have got plenty of multicast addresses in IPv6 so we can altogether avoid speculations on what the right proportion between permanently-assigned and transient addresses is: Let either category have a half of all multicast addresses. Then its category can be encoded in an IPv6 multicast address with a single bit flag, denoted **T** (Transient). In keeping with its name, the meaning of that bit will be as follows: $T = 0$ indicates a permanently-assigned address and $T = 1$ means a transient one [21, Section 2.7].

As more bit flags can later be needed in the IPv6 multicast address format, let's set aside a complete nibble of them as shown in Fig. 2.15. In fact, two other bits are also used in that nibble, known as **P** and **R**. We are going to discuss the role of the **P** flag in Section 2.9; as for the **R** flag, assume for now it is just zero.

> The meaning of the **R** flag is specified in [59]. You don't need to start reading that RFC document right away as it will be much more fun to study after we together have finished "designing" the basic components of IPv6 in the scope of this book.

The use of transient multicast addresses is managed by no central authority, and this way to tackle the issue is double-edged. On the one hand, it facilitates using multicast for everyday needs and reduces the steepness of the starting curve. Indeed, it would be too troublesome to have to request a multicast address from *IANA* every time one needs to run a small-scale multimedia conference or to experiment with a new protocol still in the works. On the other hand, conflict resolution is left up to the transient address users. Due to no registry of transient addresses available, even detecting a multicast address conflict can be a challenge and require packet-level troubleshooting. Fortunately, we already have a magic tool in our inventory that can protect from a conflict between addresses regardless of their numeric value. What we are hinting at is, of course, IPv6 address scoping, now to be applied to transient multicast addresses.

Our present focus on transient addresses doesn't mean that their permanently-assigned cousins will need no scopes. Quite the contrary, their scoping will provide for finer control over the set of interfaces a particular well-known group is to include. In IPv4 multicast, scoping was rudimentary and consisted in dedicating the block *224.0.0.0/24* to groups bearing only link-local significance. Now, with due use of the IPv6 address scoping power, it should be possible to limit any well-known group to the appropriate span. For example, assuming we deal with the Foo application protocol [62], we will need to provide means to construct at least the following multicast addresses:

- All Foo servers on this link.
- All Foo servers in this site.
- All Foo servers in the Internet.

To serve permanently-assigned as well as transient addresses, the scope of an IPv6 multicast address needs to be encoded in a clear manner within the address itself, independently of other fields. A possible recipe for that is as follows: Let there be a scope nibble immediately after the flags nibble in each IPv6 multicast address, its integer value indicating the address scope as shown in Table 2.4.

Now we are going to comment on the well-known scopes defined for IPv6 multicast addresses.

Interface-local scope will allow to limit multicasting to a single interface of the source node. Such multicast will never leave the source node and in this respect it will be similar to unicast traffic sent to the loopback address *::1*.

As we may remember, the basic IP multicast group member is an interface rather than an entire node. In other words, an IP node joins a multicast group on each interface individually rather than on all of them at once. In the multicast API terms, an interface reference such as its index or name is a

Table 2.4: IPv6 multicast address scope codes [21, Section 2.7]

Value	Scope
0	reserved
1	Interface-Local scope
2	Link-Local scope
3	reserved
4	Admin-Local scope
5	Site-Local scope
6–7	(unassigned)
8	Organization-Local scope
9–13	(unassigned)
14	Global scope
15	reserved

mandatory argument to the “Join group X” and ”Leave group X” API calls along with the group’s multicast address [51, Section 7.1][63, Section 5.2]. This is why scope 1 has its boundary drawn around a specific interface instead of the whole node.

Link-local scope is something we are already familiar with. It encompasses a single link and the interfaces connected to it.

The scopes encoded with the values 4 through 13 are intended to correspond to organizational units of different size. Unlike the smaller scopes’, the boundaries of their zones are no longer defined by the link layout or other implicit criteria. Instead, they can only be explicitly defined by the network policy. The user interface to the multicast zone configuration will be implementation-specific, but its most straightforward form can be as follows: Each zone is defined by just listing all network interfaces belonging to it.

In particular, as one would guess from their names, admin-local scope corresponds to the basic administrative domain, site-local scope encloses a single site, whatever its definition be, and organization-local scope contains a whole organization probably consisting of multiple sites. The unassigned scope codes are available for defining scopes of non-standard size, should they be needed in a particular network setup. Naturally, the greater the scope code is, the larger the scope must be.

It is worth mentioning that administratively scoped IPv4 multicast was also proposed [60].

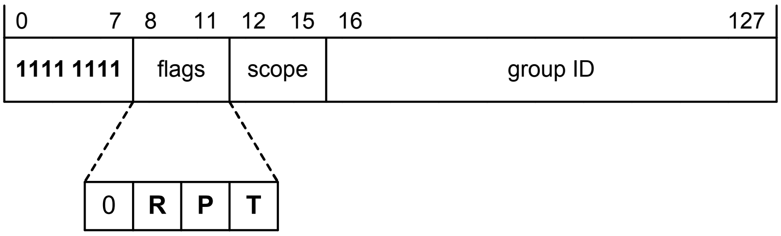


Figure 2.16: IPv6 multicast address format

Table 2.5: mDNSv6 multicast addresses of different scope

Address	Exact meaning
FF01::FB	All mDNSv6 servers on the same interface with the source
FF02::FB	All mDNSv6 servers on the same link with the source
FF05::FB	All mDNSv6 servers in the same site with the source
FF0E::FB	All mDNSv6 servers in the Internet

The largest of the well-known scopes, as one would expect, is global scope placing no spatial restriction on multicasting.

So far, we have used up 16 most significant bits in the IPv6 multicast address format. Namely, the high-order eight contain the multicast prefix *FF00::/8*, the following four were allocated to the flags nibble, and the next four are occupied by the scope code. So as many as 112 low-order bits are still available; those will make up the **group ID** field to distinguish multicast groups coexisting in a zone. The resulting IPv6 multicast address format will be as shown in Fig. 2.16.

For permanently-assigned ($T = 0$) addresses to be consistent across scopes, let well-known group IDs be assigned in all scopes at once. For instance, as soon as the "All mDNSv6 Servers" group gets an ID of `0xFB` [57], it will be a trivial task to construct the well-known addresses of different scope for that group as shown in Table 2.5 even if we don't quite have a clue what mDNSv6 is.

In practice, a well-known group may not be relevant in some scopes. For example, the "All OSPF Routers" group, *FF02::5*, is defined in link-local scope only. In that case, the same group ID is reserved in the other scopes. E.g., the multicast addresses *FF01::5* and *FF05::5* will never be assigned so as to avoid confusion.

Table 2.6: The most important IPv6 multicast groups [21, Section 2.7.1]

Address	Group
FF0x::	reserved
FF0x::1	"All Nodes"; defined for x of 1 (interface-local scope) and 2 (link-local scope)
FF0x::2	"All Routers" — defined for x of 1 (interface-local scope), 2 (link-local scope) and 5 (site-local scope)

Construct the multicast addresses of different scope for the ASAP protocol, whose well-known *decimal* group ID is 307.

As for the transient addresses ($T = 1$), they can hardly be treated the same way with respect to group ID assignment, the problem being that the admins of different levels can use the same numeric group ID for completely different purposes unless they have a common policy on that matter. Preventing a possible conflict in this case is, in a sense, what multicast address scoping is for. Thanks to scoping, a network admin can borrow a group ID in his zone of authority without having to coordinate it first with his peers, upstream providers or downstream clients. For this reason, a transient multicast address can only be relevant in a certain zone of its scope and can never suggest the meaning of the same numeric group ID elsewhere.

Notwithstanding the benefits of this scheme, only one authority can be managing transient addresses within a zone, or else a conflict will loom. In particular, it remains unclear to us who is to manage transient address assignments in the global zone. We will dedicate Section 2.9 to this issue.

And to conclude this section, let's "assign" IPv6 multicast addresses to the most important well-known groups as shown in Table 2.6.

How come the well-known groups *FF04::2* and *FF08::2* are missing from Table 2.6? They remain undefined because the "All Routers" group consists of *unicast* routers and so it is relevant only in scopes actually defined for IPv6 unicast addresses, whereas admin-local scope ($x = 4$) and organization-local scope ($x = 8$) aren't defined for IPv6 unicast. For that matter, interface-local scope ($x = 1$) isn't defined for IPv6 unicast either but, as we said earlier in this section, it is to multicast what the loopback interface is to unicast.

Another well-known assignment, of particular importance to us, is IPv6 multicast addresses for use in textbooks and documentation examples: *FF0x::DB8:0:0/96* [61, Section 3].

Unlike IPv4 multicast, whose support was recommended but yet optional [64, Section 3][51, Section 3.3.7], *IPv6 multicast is a mandatory feature.* For this reason, an IPv6 node must join the "All Nodes" group on each of its active interfaces (*FF01::1*) and the links connected to those interfaces (*FF02::1*). In addition, IPv6 routers must join the "All Routers" groups relevant to them: *FF01::2*, *FF02::2*, and *FF05::2*.

The benefits from making IPv6 multicast support mandatory will be many, but the lowest-hanging fruit to reap will be this: Any sort of broadcast can be ditched from IPv6. An IPv4 broadcast address would translate to the link-layer broadcast address such as MAC *FF-FF-FF-FF-FF-FF*, bothering the stations of the link that had nothing to do with IPv4. And now, to send a packet to the attention of all IPv6 nodes within a certain scope, the suitably scoped All Nodes multicast address is to be used. However, it won't be until Section 5.3 that we will have accumulated sufficient knowledge for the final decision to exclude any use of broadcast from IPv6. A challenge for us will be to deal with the cases in which our IPv4 experience can mislead us into summoning this terribly inefficient addressing mode.

2.9 IPv6 Multicast Addressing Extensions Based on Unicast Addresses

As we realized in Section 2.8, even scoping can't protect from multicast address conflicts when several independent users of IPv6 multicast are picking transient addresses from the same zone. Of the greatest significance is, of course, the case of the global zone. All of us and our network interfaces are in it, and so selecting global transient addresses at random will never be conflict-free, however small the actual conflict probability may be. The IPv6 multicast users will have to trust to sheer luck unless there is an international authority to manage the turnover of global transient addresses. But who is to be responsible for running such an authority and paying its bills? Nobody has stepped in so far and there seems to be little chance anybody ever will. This is the rock all attempts to set up an Internet registry for transient multicast addresses have ultimately wrecked on.

> To name just one notable attempt, **MADCAP** (Multicast Address Dynamic Client Allocation Protocol) [65] progressed as far as a Proposed Standard. However, to the best of our knowledge, it has seen only marginal use [72, Section 3.5].

What can we do about the apparently impassable obstacle we have run into? We should perhaps try passing *around* it. For instance, we can look for an existing registry suitable for piggybacking our current task on. As we are about to see, there is a perfect candidate available for this role.

Technically, the bare minimum needed for a user to connect to the IPv6 Internet is a single IPv6 global unicast address. However, to encourage people in deploying proper networks around themselves wherever they go, each IPv6 user is going to get a whole block of such addresses represented by a prefix. To handle the assignment of IPv6 unicast prefixes, there already is a global body consisting of *IANA*, several RIRs, and numerous LIRs. One of its main goals is to ensure that the prefixes so assigned are globally unique, thus preventing conflicts in the unicast address subspace.

Now let's note that it will make no sense to give anyone a prefix longer than an IPv6 subnet one, whose length was fixed at 64 bits in Section 2.6. An IPv6 subnet can't be subdivided, and so the recipient would have trouble making use of a longer prefix.

Consequently, any holder of a legitimate IPv6 global unicast prefix should be able to construct a unique multicast address based on their unicast prefix, e.g., by storing the prefix in the high-order bits of **group ID**. That field has 112 bits in it, which is more than enough to accommodate any practical IPv6 unicast prefix. No such prefix can have more than 64 bits in it, so that as many as 48 bits of **group ID** can remain available to the user, thus providing them with 2^{48} multicast addresses that are known to be globally unique!

However, a more thorough analysis reveals two problems about the trick we have just come up with:

- First, address conflicts still are possible between the two communities of users to emerge, the "new school" deriving their transient multicast addresses from their unicast prefixes and the "old school" picking them as randomly as before.

- Second, unicast prefixes can have different length and, e.g., *2001:DB8:a1ef::/48* is not the same as *2001:DB8:a1ef::/64* because the length of a prefix is its essential attribute. In addition, the unicast prefix distribution system is hierarchical, and so the owner of *2001:DB8:a1ef::/48* can be different from that of *2001:DB8:a1ef::/64* albeit the latter is downstream of the former with respect to the hierarchy.

Fortunately, neither problem is architectural, and so both of them can be solved without having to abandon the original idea. The first issue will be resolved as soon as the two communities of users are confined in non-overlapping address subspaces. This can be done with a single bit in the address flags field, denoted by **P** and already shown in Fig. 2.16 (page 76). Let the default value of **P**, which is zero, indicate the old, anarchical, way of transient address assignment, not to say appropriation. By contrast, when **P** is set to 1 in an IPv6 multicast address, we are dealing with one based on a unicast prefix. Of course, the **P** flag is meaningful as long as $T = 1$ since it is relevant only to transient addresses; otherwise **P** must be left unset, that is, zero. Here we are, the first problem solved.

The second issue is as easy to resolve. There are 48 bits of **group ID** still available and it will be no big problem if we spend a few more of them to store the prefix length, with the rest going to the multicast user as originally intended. As long as the unicast prefix length stays between 0 and 64 inclusive, 7 bits will be required to encode it as an unsigned integer. However, for the convenience of developers and users, let the field occupy a full byte, i.e., 8 bits.

The larger-than-necessary range of this field can come handy when further extensions need to be introduced. For instance, a special prefix length value of 255 marks a link-scoped multicast address derived from an interface ID instead of a unicast prefix [66]. Such an address can be generated by the node itself rather than set administratively, e.g., as may be required in Zeroconf environment.

The resulting format of a **Unicast-prefix-Based IPv6 Multicast** (**UBM**) address [67] is shown in Fig. 2.17. Although the user group ID field has somewhat shrunk from what it was in our original draft, 32 bits still are at the user's disposal—not bad at all!

We have to admit that we don't know the story how the reserved bits appeared in the UBM format, between the scope code and the prefix length. Probably they were to align the prefix field on a 32-bit boundary so that it was easier to construct UBM addresses. As all reserved fields in TCP/IP, it must be set to zero.

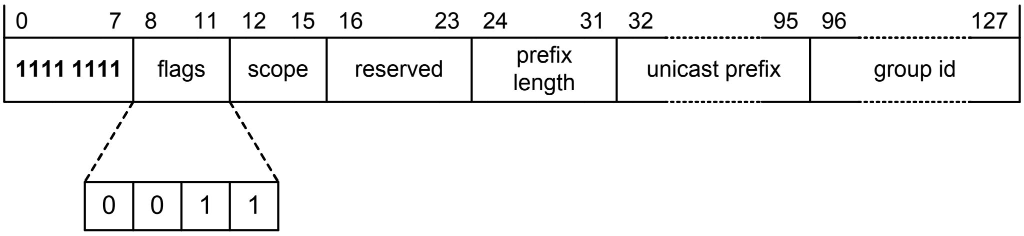


Figure 2.17: Unicast-prefix-based IPv6 multicast address format

> IPv6 isn't the only one to have UBM addresses defined; now IPv4 has got them, too [68].

For example, the owner of the global prefix *2001:DB8:D0D0::/47* will also have for their exclusive use all multicast addresses of the following pattern:

`FF3`x`:002F:2001:DB8:D0D0:0:`$zzzz$`:`$zzzz$,

where x is any valid scope code from Table 2.4 (page 75) and $zzzz$`:`$zzzz$ is an arbitrary group ID.

> A different way to say the same thing is that, along with the unicast prefix *2001:DB8:D0D0::/47*, its owner has got all of the multicast prefixes *FF3x:002F:2001:DB8:D0D0::/96* where x is a valid scope code.

> Consequently, any author "owns" all of the UBM addresses based on the IPv6 unicast documentation prefix *2001:DB8::/32* and its longer subprefixes [61, Section 3.1].

Thanks to a convenient field alignment, all parts of this UBM address can easily be made out in its text format. In particular, `002F` is just the prefix length value (47) in hex and it is immediately followed by the prefix proper. An interesting feature of a UBM address is that it will be easy to track it back to its unicast network of origin although the group itself can have global coverage and consist of interfaces in many networks all over the world.

The final touch to give to the UBM scheme is, of course, about scoping. Will it be possible to base a UBM address on a non-global unicast prefix? Suppose for example that it is a site-local prefix. Where did its owner get it from? Apparently, that user is with a small or mid-size organization, or with a part of a larger organization, where the network is site-local address based, essentially being a site-local zone. That zone is also expected to have an address-assigning authority, such as a network admin or an IT department. So it was that authority that assigned the site-local prefix to the user, and it was in the authority's best interests to ensure that the prefix would be unique within the zone. Therefore a UBM address based on the site-local prefix will be as unique within the same zone. To generalize, a UBM address based on a scoped prefix will be unique and hence valid only within the scope of that prefix. Therefore *the scope of a unicast-prefix-based multicast address must not be larger than that of the unicast prefix used* [67, Section 4]. For instance, the user having a site-local unicast prefix in possession may construct interface-local, link-local, admin-local, and site-local UBM addresses from it; but he may try to construct neither organization-local nor global UBM addresses this way.

> A task for zone-border multicast routers will be to deny packets whose destination address violates the UBM scoping rule.

Having finished with the UBM way of ensuring the uniqueness of transient multicast addresses, let's see if there are any significant, if specialized, alternatives to it. One important scenario is when multicast traffic can be bound to a specific source. For example, such are cases of TV-like or radio-like multicasting when a so-called **channel** has a single, predefined source while the number of listeners can vary from zero to infinity as usual. This mode is quite different from the traditional IP multicasting scheme where any source can be sending traffic to the given group of listeners.

When multicasting from a specific source, that source is identified by its IP address. Should a different source be sending packets to the same multicast destination, that will be considered a different channel. Therefore a channel is identified by the tuple (S, G) where S is the unicast source address and G is the multicast destination address. E.g., to check if an incoming multicast packet is actually addressed to it, a listener in this mode will be using the list of channels (S, G) instead of the list of groups G joined on the arrival interface of the packet. This multicast mode is known simply as **Source-Specific Multicast** (**SSM**) [69].

In the same vein, the traditional mode where any source can be multicasting to a group is referred to as **Any-Source Multicast** (**ASM**) [70, Section 2].

Does SSM addressing exhibit any unusual properties? In SSM, it is the channel address (S, G) that is to be unique within the relevant zone. In other words, channels (S_1, G_1) and (S_2, G_2) will be the same if and only if $S_1 = S_2$

and $G_1 = G_2$. Conversely, different sources S_1 and S_2 (i.e., $S_1 \neq S_2$) will never be able to send to the same channel even if they are sending to the same multicast destination address G. Consequently, an SSM source doesn't have to be concerned at all about the uniqueness of the multicast address G as long as its own source address S is unique. Distinct multicast addresses G_1, G_2, and so on will only be required for different channels originating from the same source S.

Between the two extremes of ASM and SSM lies a hybrid multicast mode where listeners accept packets only from a specific set of sources rather than from all of them. So it is known as **Source-Filtered Multicast** (**SFM**) [70, Section 2]. In its simplest form, SFM can be implemented as a packet filter in the listener's network stack. However, the multicast efficiency considerations dictate that the network, too, participate in filtering multicast traffic and prune the paths to where nobody is listening to a particular group or channel. That's why SFM support is found in the modern IP multicast management protocols IGMPv3 and MLDv2, the latter to be introduced in Section 6.4. By its addressing, SFM is a much closer relative to ASM than to SSM because its groups are identified only by the multicast destination address G, which needs to be unique, while the source address S is used only for filtering at a later stage.

A hidden issue in our discourse is that we have spoken of the multicast addressing types as of *modes*. Will there be a manually operated switch in each IPv6 implementation for selecting ASM mode or SSM mode? Having it that way would be awkward and, for that matter, do little credit to our protocol design skills. Instead of relying on a system-wide switch forcing certain mode of operation, the IPv6 stack should be able to tell if an incoming packet was sent in ASM mode or SSM mode and handle it appropriately.[13] Ditto for the network trying to optimize its multicast traffic. Then the source will be able to apply this or that mode to each multicast transmission as requested by the application. For this beautiful plan to come true, let the distinction between ASM and SSM be encoded in the multicast destination address G. In other words, any destination address G to appear in an SSM channel needs a bit-level feature that can set it apart from regular ASM addresses.

It can be speculated that SSM channels (S, G) are akin to UBM addresses in that a unicast address or prefix appears as an integral part of the extended multicast address. The difference between the two is only this: In the UBM

[13] This will be in accord with the architectural principles of the Internet [71, Section 3.8].

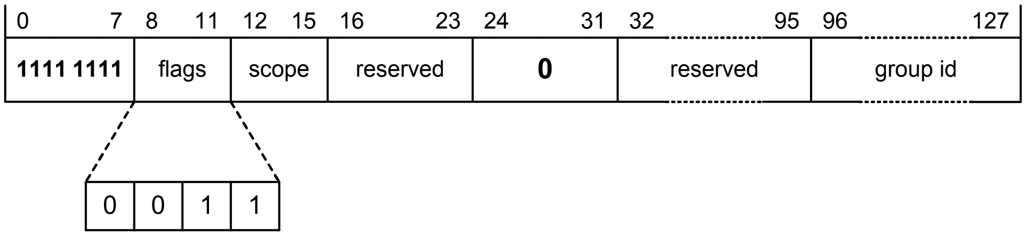


Figure 2.18: IPv6 source-specific-multicast address format

format a unicast prefix is encapsulated inside a 128-bit IPv6 multicast address whereas in an SSM channel a unicast source address and a multicast destination address remain distinct and have to come side by side, e.g., as they would do in the IPv6 header we are to work on in Section 3.2. This observation suggests that a recognizable format for SSM destination addresses can be made just a special case of the UBM format from Fig. 2.17.

Putting ourselves in *IANA*'s shoes, let the empty prefix *::/0* be dedicated to SSM addresses.[14] Namely, an IPv6 multicast address with $T = 1$ and $P = 1$ and a UBM prefix length of zero, as shown in Fig. 2.18, will be identified as an SSM address. It isn't difficult to work out that all SSM addresses are exclusively represented by the *FF3x::/96* prefixes where x is a valid scope code from Table 2.4. So there are 2^{32} addresses available in each scope, which can hardly be regarded as a large number today. It shouldn't be a problem though since the 2^{32} addresses are per SSM source rather than for the entire Internet. All those addresses can be used by each source out there at the same time because the source address will be part of the complete channel ID.

In fact, the SSM idea isn't really IPv6-specific; only its implementation is. So it is no surprise that SSM has also been defined, mutatis mutandis, for IPv4 [69]. IPv4 SSM addresses come from the *232.0.0.0/8* block [58].

The last issue to touch upon while at SSM will be our customary one, that regarding scoped addresses. In any SSM channel (S, G), its components S and G are mere IPv6 addresses, each having a scope and, when in the context of a network, belonging to a zone of that scope. Will new rules be required to make S and G agree with respect to their scopes and zones? In fact, no; the most general rule based on the zone isolation principle from Section 2.4 will be sufficient: The source interface bearing address S must

[14] *IANA* would have right to do that because *::/0* is theirs.

be in the zone of address G and all interfaces forming group G must be in the zone of address S.

To conclude this section, we need to point out that the **group ID** value in any transient multicast address ($T = 1$), including UBM and SSM, is restricted to the range of `0x8000 0000` through `0xFFFF FFFF` by the multicast address allocation policy [73, Section 4.3]. The rationale behind this decision will become clear after we discuss in Section 4.1.1 how IPv6 multicast addresses are to be resolved to MAC-48 addresses.

2.10 Site-Local Address Problem and Its Solution

The final IPv6 addressing issue we are going to consider may seem surprising at first. Here it is: What shall we do about the site-local addresses after all? But hang on—it was nobody else but ourselves who introduced those addresses in Section 2.3 and extensively used them in our examples whenever we needed addresses of different scope, was it not? Yes, it was, and we aren't going to deny that.

The only our problem was that we relied on an idea of a site that didn't have enough in common with the experience of everyday network operation. Our perfect site would be an area with clearly-defined and never-changing borders, whereas in practice both qualities of a site, its distinctness and its immutability, are being constantly challenged by the reality. The most demanding of the challenges perhaps is site merger. Suppose two organizations with their networks based on site-local addresses want to merge. It will be natural for them to link their networks up as well. If they are exceptionally lucky, their networks won't have a single subnet prefix in common. Then the addresses already assigned will remain unique and the only setting to review will be the zone indexes in the former border routers so that the two sites become one; regular IP routing will be able to cope with the rest. But, notwithstanding the IPv6 site-local address block being as huge as a */10*, the two sites are too likely to end up in address conflict due to human factor. Guess which prefixes network admins prefer? Right, they like prefixes that have a more compact spelling and thus are easier on the admins' typing fingers and memory, such as *FEC0:1::/48*, *FEC0:0:0:1::/64*, etc. For this reason, a merger of two large sites will almost inevitably result in some conflicts in the joint address plan.

Anyone who has had the unforgettable experience of merging two or more *10.0.0.0/8* private IPv4 networks can confirm what we just said here about the role of human factor.

As an exercise, the reader will spot a few more practical scenarios where the use of site-local addresses will cause more problems than it is supposed to solve.

Thus we arrive at the seemingly odd conclusion that private-use addresses had better be globally unique. Of course, they still are to be kept out of the Internet but their newly acquired uniqueness will greatly facilitate the interaction between different sites. For example, a community of private networks will be able to get interconnected and share their network resources by mutual agreement, building a network of networks, a "private Internet", if you will. An undertaking of this kind will, no doubt, require hard work in other areas besides TCP/IP, such as politics and management, but at least it will meet no obstacles as far as IPv6 addressing is concerned.

Solving this problem is easy thanks to the IPv6 address having enough bits in it for various tricks. By our plan from Section 2.1, each site should be able to get a */48* prefix. If 8 of those bits are occupied by a well-known prefix common to all private-use addresses, 40 bits will still be available for a unique site ID. What will the conflict probability be if N sites pick the 40 bits of their ID at random? It isn't difficult to recognize this problem as based on the widely known birthday paradox. What is paradoxical about it is that the result will be considerably higher than the probability to hit a preselected value, the latter being as low as 2^{-40} in our case.

To approximate the conflict probability, there are a few elementary formulae known [74], one of them shown in Equation 2.1. In the formula, N is the number of sites and M is the number of alternatives a site can pick from. What this formula gives is the probability of a conflict between just two sites. The probabilities of the higher-order conflicts are significantly lower and can be just disregarded.

$$p \approx 1 - \left(1 - \frac{N}{2M}\right)^{N-1} \tag{2.1}$$

In our case, $M = 2^{40}$ and N can be varied as shown in Table 2.7 for us to get insight into the matter. Are those probabilities sufficiently small? If true global uniqueness is the goal, then they are probably not. E.g., a million sites will suffer a conflict with a probability of about one third. However, that conflict will actually manifest itself only if the million of sites decide

Table 2.7: 40-bit site ID conflict probability

N	p
2	9.09×10^{-13}
10	4.09×10^{-11}
10^2	4.50×10^{-9}
10^3	4.54×10^{-7}
10^4	4.55×10^{-5}
10^5	4.54×10^{-3}
10^6	0.365
10^7	> 0.999

to link up into a single entity, which is extremely unlikely.[15] Thanks to this observation, we can better understand what our actual goal is. We are by no means trying to provide for building an alternative internet based on private-use addresses. All we want is to ensure that the interaction of a *reasonable* number of sites will, with sufficient probability, be conflict-free. We still need to define though what number of sites is reasonable and what probability is sufficient. It is an undebatable truth that the more sites are to be interconnected, the greater the management overhead will be. Therefore, should there be a real lot of sites to interconnect, say, 100 or 1000 of them, they can be better off using public IPv6 addresses and interacting via the Internet, where all the administrative issues have already been sorted out and the required mechanisms created, rather than trying to reinvent the wheel with private-use addresses. So it can be speculated that 1000 sites is an absolute limit; and the conflict probability for $N = 1000$ still is as small as one in two million.

Whether a conflict probability of $1 : 2,000,000$ is sufficiently low, the reader can decide themselves. To put this value into perspective, the probability that the flight you have boarded will crash and you will die in the crash is of the same order of magnitude [75].

We will accept this result as satisfactory and agree that addresses based on a prefix with 40 random bits in it meet our requirements. Now we just need to give them a name and specify their format details.

An IPv6 address based on a random prefix is known as a **Unique Local Address** (**ULA**) [76]. The prefix clash probability still greater than zero

[15] We leave it up to the reader to calculate that unlikeliness if they feel like doing so.

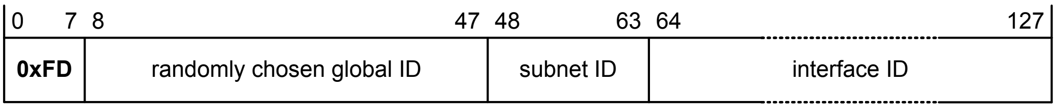


Figure 2.19: Unique local IPv6 unicast address format

makes these addresses only quasi-unique but that will be enough for the purpose of ULA.

The technical details of ULA are as follows [76, Section 3]. The 8-bit well-known prefix included in each of them is *FD00::/8*. It is followed by 40 random bits known as the global ID. The well-known prefix and the global ID constitute a */48* prefix for the site to use. The next 16 bits are available to the site for subnetting, each subnet being a */64*. The resulting format is shown in Fig. 2.19. For example, should the random ID ever come out at `0x123456789A`, the complete prefix for the site to use will be this: *FD12:3456:789A::/48*. Of course, in reality a crypto-grade random number generator must be used to get a global ID so as to minimize conflict probability.

What scope will ULA have? The whole point of introducing ULA was to let private networks develop freely and communicate with their peers as soon as they need that. So as to avoid introducing artificial obstacles in the way, ULA are granted with global scope by default and they will share one global zone with public IPv6 addresses. That said, ULA are prohibited from appearing on the public Internet because they remain unique only as long as a limited number of private networks meet up.

The adjacent block, *FC00::/8*, is also reserved by the ULA RFC [76]. Its addresses are to have the same ULA format: an 8-bit well-known prefix followed by a 40-bit global ID followed by a 16-bit subnet ID followed by a 64-bit interface ID. However, global ID assignment is to be centrally managed in this block using a mechanism yet to be defined, e.g., by a registry [77]. Together the two ULA blocks occupy the prefix *FC00::/7*.

Discuss the applicability scope of those Unicast-Based Multicast Addresses from Section 2.9 that are derived from a ULA.

Thanks to ULA, it is now possible to eliminate the use of site-local addresses from the IPv6 practice. In fact, site-local addresses were deprecated by *IETF* a while back [36, Section 4], but only a viable alternative can motivate the IPv6 users to actually comply with that decision.

2.11 IPv6 Addresses in DNS

Despite the shorthand notation, IPv6 addresses can be significantly harder to remember than host names. Furthermore, a host name can be handier than a numeric address in that it is supposed to be stable and never change, at least as long as the host retains its current role.

> The maritime network operators even believe that renaming a host can bring nothing but bad luck.

The place to keep the mappings between host names and IP addresses is, of course, DNS. What will it take to make DNS IPv6-ready as far as publishing IPv6 addresses in it is concerned?

The first detail to pay attention to is the forward zone. The IPv4-specific mappings were stored in it as *A* resource records (RRs). Such a record's **RDATA** format was totally inflexible, consisting of just a 32-bit field to put an IPv4 address in. Because of that feature, an *A* record could have nothing but an IPv4 address on the right-hand side. To store an IPv6 address there instead, a new RR type will have to be introduced.

Assuming IPv6 is going to be the ultimate Layer 3 protocol, the new RR type's **RDATA** won't need much flexibility either: It will be just an IPv6 address stored as a 128-bit binary value in network order [37, Section 2.2]. The IPv6 address being four times as long as its IPv4 predecessor, the new RR type will have this ingenious name: *AAAA*, pronounced "quad-A" [37, Section 2.1]. Its well-known code value will be 28.

> Of course, we upheld the inflexibility of the *AAAA* RR with tongue in cheek. It is hardly a good style for a protocol designer to introduce a new record type for each address family. An extra field to indicate the address family within a generic, protocol-neutral record format would be a more elegant solution. However, flexibility lost the *AAAA* case to simplicity.

The text form of a *AAAA* record to appear in a master file is obvious: On the left is the domain name and on the right is the IPv6 address in the standard notation from Section 2.2. For example:

```
www.example.org.     AAAA    2001:db8:c001::f00d
```

Excellent, the forward zone is now IPv6-ready and we can move on to the reverse zone, which keeps the mappings from IP addresses to host names.

PTR records are employed for that purpose, and they are entirely protocol-agnostic. For that matter, those records have little to do with addresses as such because their **RDATA** are domain names [38, Section 3.3.12]. Anyway, now we need to store an IPv6 address on the *left-hand* side of such a record, where any DNS record would have a domain name.

To disguise an IPv4 address as a domain name when in the same situation, the following tools were required: the reverse byte-wise notation and the well-known domain *in-addr.arpa*. The reverse notation provided for delegation of IPv4 prefixes as subdomains. IPv6 will need this feature too, but let's reduce the delegation quantum from the byte to the nibble so that finer prefixes can be sub-delegated. In addition, so as not to confuse the two IP versions, a separate well-known domain will be used for IPv6 reverse mappings: *ip6.arpa* [37, Section 2.5].

Each nibble of an IPv6 address becomes a hex digit of its text notation if fully expanded, that is, with all of the zeros spelled out. So the *PTR* encoding procedure for an IPv6 address can be as simple as this: First, undo the "`::`" contraction if present and spell out the leading zeros, if any, in each group so that the address has 32 hex digits in it. Second, write the address digits back to front, each followed by a dot character. Finally, append the *ip6.arpa* domain from the right. That's it!

For example, the above address *2001:db8:c001::f00d* needs to be expanded as follows:

```
2001:0db8:c001:0000:0000:0000:0000:f00d
```

Then converting it to a reverse domain name will be straightforward, if you excuse this oxymoron:

```
d.0.0.f.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.0.c.8.b.d.0.1.0.0.2.ip6.arpa.
```

There will be 32 further levels under *ip6.arpa*—just as many as there are nibbles in an IPv6 address. With no shorthand notation available for domain names, all levels have to be spelled out.

> Fortunately, when populating the actual master file for a zone, the common domain suffix can be specified once in an `$ORIGIN` entry rather than repeated all over the place [38, Section 5.1].

And now we should raise our favorite question about the coexistence of whatever we have just "created" and the IPv6 scoped address architecture from Section 2.4. In particular, will entering IPv6 scoped addresses in DNS make sense? We won't be able to provide a complete answer to this until Section 6.7, when we have all keys to the issue. It is already clear to us nonetheless that *non-global IPv6 addresses shouldn't be in public DNS zones.*

> This rule also applies to the well-known addresses such as the loopback address, *::1*. Should global names ever need to be assigned to them, that will be up to the Internet authorities, *IANA* and *ICANN*, rather than to each DNS operator.

Additionally, even in a private DNS, scoped addresses can't be extended with a zone_id because the latter is meaningful to one node only whereas DNS provides its information to many nodes. It would've been pointless to publish IPv6 addresses with zone_id in DNS, and so we were right to allocate just 128 bits to the *AAAA* **RDATA**, thus disregarding any possible zone_id. Whenever a scoped IPv6 address is learned from DNS, its scope zone should be clear from the context. E.g., site-local addresses are supposed to belong to the site that has published them on its private DNS server; naturally, they shouldn't be visible to other sites, or confusion will ensue.

> Take note that "zone" is an overloaded term in TCP/IP: The IPv6 scope zone and the DNS zone are completely different things, not to be confused.

Chapter 3

IPv6 Packet

3.1 IPv6 Packet Layout

Now that the basic questions of IPv6 addressing have been answered, we can move on to the next key element of IPv6: its packet.

> Notwithstanding our revolutionary spirit, we aren't going to question the basic concept of the network packet: IP will always be a packet switching protocol.

The well-known packet layout consisting of a header and a payload section may not be optimal in every possible case out there, but it certainly is one of the simplest trade-offs possible and everyone seems happy with it. So we can avoid reinventing the wheel and just once again adopt this packet layout in its most general form as shown in Fig. 3.1.

This decision wasn't a hard one to make, but the following issue is more challenging: How can the header be made flexible and extensible? Without these two qualities, no doubt, the protocol won't last long. IPv4 had options for this purpose, and they more or less did the job. Why should we not be satisfied with them though?

Firstly, IPv4 options lacked a framework governing their order. In general, there can be two flavors of IP options, one to be read and acted on by

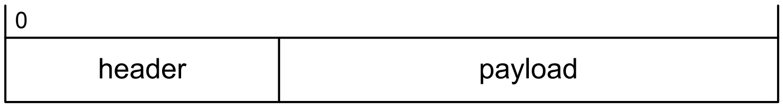


Figure 3.1: Our first approximation to the IPv6 packet structure looks rather traditional

each node in the packet's path and the other carrying instructions for the destination only. Let's refer to the former as **hop-by-hop options** and to the latter as **destination options**. In case these two option types are allowed to alternate in the packet header, a router will have to scan through their entire list to pick out all hop-by-hop options, the extra work slowing down packet forwarding.

As a matter of fact, an IPv4 packet could have more than one destination node. E.g., a multicast packet would be addressed to a group of nodes, each of them effectively being its destination. Apart from that, each address listed in a source-route option, LSRR or SSRR, counted as a destination although only the last one in the list would be final. Both cases prove as applicable to IPv6: Multicast addresses were already introduced in Section 2.3 and Section 2.8 while IPv6 source routing will make its appearance in Section 3.3.3, where different destination types will also be commented on.

Secondly, it goes without saying that end-to-end IP security is a mandatory feature to have, and it requires that the information contained in a packet be protected directly at IP level rather than by a lower- or upper-layer protocol. Consequently, auxiliary security data such as a digital signature has to be part of the IP header or its extension.

By IP security, we mean authentication, encryption, and integrity assurance of IP packets. In the TCP/IP stack, this task is handled by the IPsec family of protocols, to be briefly touched on in Section 3.3.5.

The end-to-end quality to IP security means that it only concerns the source and the destination while being completely transparent to intermediate nodes such as routers: The latter are agnostic to the protection status of individual packets and forward all of them using the same procedure.

Suppose our means were limited to IP options only; then we would have to implement IP security components as options, too. At the same time, the destination options already present in the packet could benefit from encryption since no router would be inspecting them. However, their selective encryption would leave the packet header with some of its sections in cleartext and the others encrypted, to a great confusion of the routers to forward the packet. To avoid confusing the routers, the destination options would need to be required to come after the security options. In that case, the routers would need to recognize security options in order to know when to

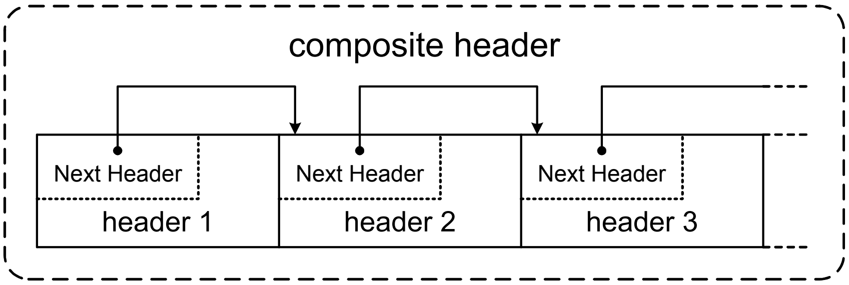


Figure 3.2: The composite header of an IPv6 packet

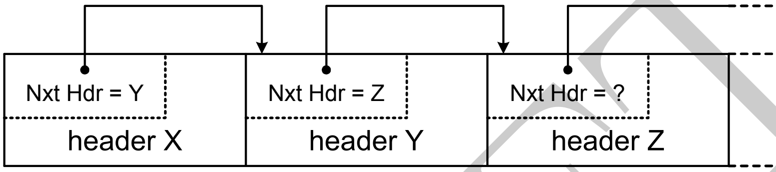


Figure 3.3: The type of a header is to be looked up in the preceding header.

stop scanning through the packet header so as not to run off into the ciphertext gibberish. Apart from this scheme's sheer inelegance, it would also compromise the end-to-endness of IP security by exposing it to intermediate nodes. If we had only IP options in our toolkit, we would find ourselves in trouble!

Fortunately, we aren't really limited to IP options as they were in IPv4: We are free to revise the IPv6 packet header structure so that it becomes more than just a fixed part followed by some options. Let it be truly modular and consist of small and simple headers, each accomplishing one specific task. For the chain of headers to have a virtually unlimited length, they can be linked in a way having something in common with linked lists and TLV encoding. Namely, each header will have a **Next Header** field indicating the type of the header to follow as shown in Fig. 3.2 [82, Section 4]. For this scheme to work, variable-length headers will also have to include a length field, while fixed-length headers won't because their length will have been nailed down in the protocol specification. The length information will allow the network stack code to locate the end of the current header and move focus to the next one.

One might say that the IPv6 packet header owes its flexibility and extensibility to recursive encapsulation.

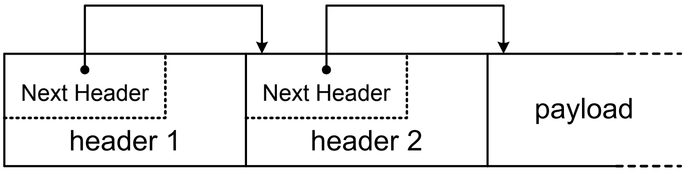


Figure 3.4: How the composite header is linked with payload

Note well now: Unlike in regular TLV encoding, where an object's type is stored in the object itself, the type of an IPv6 header is stored in the preceding header as shown in Fig. 3.3. This format provides the receiver's stack with no clue regarding the type of the very first header in the chain. However, that won't be much of a problem if the first header always has the same predefined type. A mandatory header, to be carried by every IPv6 packet, will be required anyway in order to contain the most basic information about the packet such as its source and destination addresses; so it makes perfect sense to put it in front of the chain. By a quirk of terminology, this opening header is known as *the* **IPv6 header** while the other headers are referred to as **IPv6 extension headers**. When there may be ambiguity, we will be saying, "the main IPv6 header", to mean the basic header every IPv6 packet starts with.

The composite IPv6 packet header made up of simple headers has no well-known name because it is normally considered as just a sequence of simple headers rather than an entity on its own.

And what Next Header value is to be stored in the last header? It will be consistent with the composite header design to put the payload type in there because the payload is what follows the last header. However, IP payload types have already got a registry with *IANA* [83]. To harmonize the design back with the practice already in use, IPv6 header types need to come from the same registry; then it will be OK to terminate the header chain in any header as shown in Fig. 3.4 because the **Next Header** field becomes indistinguishable from the payload type, aka **Protocol**, field. Otherwise we would have to introduce a special extension header to contain payload type *instead* of **Next Header**, but the unification of the payload and next header type registries makes it redundant.

This registry choice dictates the length of the **Next Header** field: It will be 8 bits because there are 256 distinct values in the Internet protocol registry. As it can be expected, the small size of the registry will be limiting the number of different header types that can appear in IPv6.

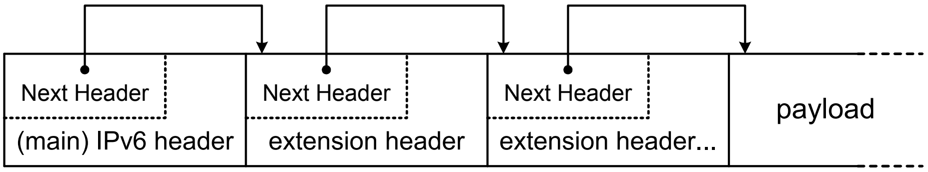


Figure 3.5: The final structure of the IPv6 packet (the number of extension headers is variable)

We deliberately said just, “header types”, rather than, “extension header types”, here. The main IPv6 header has got a distinct code assigned from that registry, too: 41. It will be of use for tunnel encapsulation as discussed in Section 4.2.

Fortunately, almost half of the values in the Internet protocol registry are still free.

We will see in Section 3.3.5 that IPsec headers, by their design, are just IPv6 extension headers. If so, how can they apply to IPv4? It isn’t hard to see that it would be perfectly possible to attach a chain of extension headers after the conventional IPv4 header because the protocol registry is the same regardless of the IP version. This trick’s practical use is very limited though because most extension headers are only relevant to IPv6 and would be completely meaningless if in an IPv4 packet, the IPsec headers being a notable exception.

To conclude this section, let’s sum up all of our decisions regarding the IPv6 packet structure in a single illustration: Fig. 3.5.

3.2 IPv6 Header

So each IPv6 packet still starts with a basic header—surprise! However, unlike its IPv4 cousin, the IPv6 header can be stripped of all optional elements, to be put in extension headers. What fields are a must in the main IPv6 header? Based on our IPv4 experience and keeping in mind what we said in Chapter 1 and Section 3.1, we can compile the following, tentative and unordered, list:

- Version;

- Source Address;
- Destination Address;
- TTL;
- Packet Length;
- Next Header.

Now let's review each field and adjust its properties if necessary.

> Integer fields we are going to meet herein will be unsigned by default. Why do you think signed and floating-point fields are rarely encountered in network protocols?

As was discussed in Chapter 1, a **Version** field will allow network stack implementations to distinguish IPv6 packets from IPv4 packets as long as both versions are present in the Internet. IPv4 is extremely widely deployed and the investments in IPv4-based technologies are huge, so the transition to IPv6 will go on for years if not decades. To play its role, **Version** needs to occupy exactly the same spot as in the IPv4 header, so it will be 4 bits long and located at the very beginning of the IPv6 header, with respect to network bit order. What will be different is the version value: 6.

We also can't do without two IPv6 addresses in the header, Source and Destination. They are going to occupy as many as 256 bits, or 32 bytes, in total, which isn't quite peanuts. Should we try to compress their binary form with some encoding, like we already did to the text notation in Section 2.2? No, it would be an unwise thing to do because, for a few bytes' gain, it would sacrifice the header's simplicity and impede its processing, particularly in switching hardware. What rules need to be specified regarding those addresses? First and foremost, *a packet never has more than one source interface and so it must not have a multicast Source Address.* Should a packet with a multicast Source Address be received, it must be immediately discarded with no attempt to react to it because any response to it will open up an opportunity for an amplified DoS attack on other nodes. Apart from that, the loopback address *::1* can appear only in loopback traffic and must never come from the wire, be it Source or Destination. We will work out other requirements to the addresses in the IPv6 header in due course.

> By the way, it is thanks to the IP Source Address never being multicast that we can attribute a type to the whole IP packet based only on its Destination Address flavor, as in "a unicast packet" or "a multicast packet".

Having no TTL field at all would give rise to the risk of packets looping in the network infinitely. Under unfavorable conditions, packet looping can be accompanied by packet duplication, resulting in an exponential burst of stray traffic and ultimately a network meltdown. So an equivalent of TTL must be included in the IPv6 header. Still, we are free to adjust its name and semantics to the established practice of putting an upper limit on the number of hops rather than on the packet's lifetime as such. As we know, the original IPv4 specification required that TTL count down seconds spent by the packet in transit and each router decrement it by the number of seconds the packet had been held for, rounding any sub-second interval up to one second. However, in practice a packet would very seldom be held by one router for more than a second, effectively turning TTL into a mere hop count. In addition, no router vendor could be bothered to actually keep track of the packet hold time. So the time limit function was eventually dropped from the standard IPv4 TTL semantics [84, Section 5.3.1], redering "TTL" a misnomer. Taking this into account, the IPv6 header will have a **Hop Limit** field instead. Unlike the field's name, its length will remain the same: 8 bits. By its meaning, a **Hop Limit** value can be neither negative nor fractional, so it can always be stored as an unsigned integer.

> The original intention behind counting down IPv4 packet lifetime, in seconds, was to control timers in the destination host as well. For example, in the original IPv4 reassembly procedure, the reassembly timeout would be set on a fragmented packet based on its residual TTL value [85, Section 3.2]. However, this approach was later abandoned in favor of robuster ways of timeout management. In particular, it was recommended that the IPv4 reassembly timeout value be just fixed [51, Section 3.3.2].

The rules of handling **Hop Limit** are just the same as the de facto rules for the IPv4 TTL:

- The packet source initializes **Hop Limit** to a value of its liking within the field's range.
- Each intermediate node decrements **Hop Limit** and, should its value become zero, discards the packet and probably sends an equivalent of an ICMP message back to the source.
- The final destination disregards the **Hop Limit** value.

These apparently simple rules may still need a few in-depth comments:

- First, if the source is going to send the packet on the wire, the packet is bound to take at least one hop through the network. Therefore a packet should never be originated with a **Hop Limit** value of zero.

- Second, a node is not to work on **Hop Limit** in an incoming packet until it makes sure the packet is to be forwarded to elsewhere. If the node is the final destination, it must never discard the packet based on its **Hop Limit** value.

- Third, in reality a packet can be received from the wire with a zero **Hop Limit** resulting from a fault in the previous hop or a malicious attack. Decrementing it as an 8-bit unsigned integer will yield 255 and so the offending packet will be free to make 255 more hops instead of being immediately discarded. Because of this quirk, **Hop Limit** needs to be checked before getting decremented: Should its value be one or zero, the packet must go into the bit bucket.

The reference IPv6 implementation from *KAME* [86] works around the case of a zero **Hop Limit** by comparing the field's value with 1 *before* the decrement instead of 0 after it; if the value is 1 or less, the packet will be dropped. The algorithm proposed by the standard [82, Section 4.4] relies on the same trick to handle this corner case. However, the text of the standard [82, Section 3] could have done a better job drawing attention to this singularity and stressing the importance of handling it right.

A Packet Length field is necessary to provide for link-layer protocols unable to keep track of the exact payload length. The now ubiquitous *Ethernet* is one case of such a limitation at link level: Its payload length can't be less than 46 bytes due to its roots in CSMA/CD, and packets shorter than that have to be padded to meet this requirement. Another, somewhat different, case can arise if the link-layer protocol requires that its payload length be a multiple of, say, 8 bytes.

The latter limitation is rather typical to a secure protocol based on a block cipher, but it isn't always exposed to the consumer, i.e., the upper-layer protocol. E.g., the PPP encryption subsystem conceals it for the consumer's convenience by means of a special Self Describing Padding appended to the payload and carrying the information of how many padding bytes were appended [87, Section 6.1]. Thanks to this information, the padding can be cleanly removed by the PPP receiver.

The path of a packet through the dynamic and heterogeneous network is unknown beforehand and, in theory, the packet may have to travel through a link of any possible type. That's why a Packet Length field or its equivalent will have to be present in every IPv6 packet's header. The length unit will be the same as in IPv4, the byte, and the packet will consist of an integer number of bytes simply because now it is convenient to have it this way, for all information processing technology has come to be byte-oriented.

> Historically, there were conflicting definitions of the byte and so the term "octet" was introduced to have the specific meaning of an 8-bit unit. Fortunately, today the byte is no longer ambiguous either, meaning exactly the same thing as the octet. We prefer "byte" because it is familiar to a wider audience while "octet" is gradually slipping into oblivion except in a few special cases such as French.

An unsigned 16-bit field should be enough to store the IPv6 packet length because, on the one hand, the link-layer protocols in common use still provide no support for MTU greater than $\sim 2^{16}$ bytes and, on the other hand, the existing transport protocols such as TCP and UDP have no use for IP packets longer than $\sim 2^{16}$ bytes. There is a good reason to keep the packet length relatively small: The longer the packet, the greater the probability it gets corrupted in transit. It is what a packet switched network is about: transmitting even a huge chunk of data in packets of reasonable size for efficiency's sake.

> How do TCP and UDP limit the usable packet size? The UDP case is trivial: There is a 16-bit Length field in the UDP header. By contrast, TCP stores no segment length, relying on IP to do the right thing. At the same time, the TCP Urgent Pointer is a 16-bit field, thus limiting the span of urgent data to $\sim 2^{16}$ bytes into the segment. And more importantly, a TCP Maximum Segment Size can't be over 65535 bytes as limited by the MSS option format.

> Nevertheless, there is a Jumbogram option to extend the IPv6 packet length to a 32-bit value [88]. No doubt, TCP and UDP would have to be extended as well to make use of a larger packet. The *KAME* IPv6 stack supports the Jumbogram option just in case.

An IPv6 packet may contain options but it doesn't need to, so it makes sense to store IPv6 options in an extension header, to appear in the packet only as necessary, rather than clutter the main IPv6 header with them. Thanks to this move, our prototype of the IPv6 header ends up with a constant number of fields in it and a fixed total length.

So far, as many as 36.5 bytes of the IPv6 header have been occupied by absolutely essential fields. Now let's align its end on a nice word boundary to make life a bit easier for silicon chip designers and network stack developers. Unable to shrink the essential header even by a nibble, we have to beef it up to 40 bytes, providing for a nice 8-byte alignment 64-bit platforms can benefit from. Owing to the alignment, we get 3.5 bytes extra to find a use for.

We are going to yield to the fashion for QoS and use the extra bits for a **Traffic Class** (8 bits) and a **Flow Label** (20 bits). The former will be analogous to the IPv4 TOS and the latter will allow multiple packets to become one entity for policy purposes, a **flow**, provided that they are marked with the same **Flow Label** value. We will delve no further into the application of these two fields.

As we know, today the IPv4 TOS is to be interpreted according to DiffServ [89] and ECN [90]. The same will apply to the IPv6 **Traffic Class**.

A general guideline for handling **Flow Label** is set in [91]. A promising application of **Flow Label** can be in load balancing across links [92] and servers [93].

We don't mean that QoS is just a fad or snake oil. However, too often does it get deployed in a network out of mere fashion, with no proper understanding of its reasons, rules, and workings. Ironically, opponents of QoS often are as naive.

Now that we have worked out the total length of the main IPv6 header and it has proved to be a constant value of 40 bytes, this number can be deducted from the value of the tentative Packet Length to increase it is range. A 16-bit unsigned field, it has a range of 0–65535, but including the IPv6 header length in it would reduce its effective range to 40–65535 because the IPv6 header is to be found in every IPv6 packet. By adjusting it down by the constant header length, the total packet length range can be extended to 40–65575 bytes. Of course, in this case the length field semantics is going to change as it will no more be a straightforward Packet Length. For the name to match the purpose, let's rename the field to **Payload Length**. In this context, “payload” is whatever follows the main IPv6 header, including any extension headers. By what we just said, the maximum possible length of such “payload” is 65535 bytes and that of the whole IPv6 packet is 40 bytes greater, i.e., 65575 bytes. This is a theoretical limit stemming from the way the packet length information is encoded in the IPv6 header; there will also be encountered practical limits, to be discussed along with IPv6 fragmentation in Section 3.3.4.

> When scanning through the chain of headers in an incoming packet, an IPv6 stack implementation must be doing boundary checks to ensure that the chain doesn’t run off the packet’s end. It goes without saying that each element of a packet must lie within the packet’s boundaries; otherwise the packet is obviously malformed and can crash the stack if processed unawares.

Now a complete, if unordered, list of IPv6 header fields can be compiled:

- **Version** (4 bits);
- **Source Address** (128 bits);
- **Destination Address** (128 bits);
- **Hop Limit** (8 bits);
- **Payload Length** (16 bits);
- **Next Header** (8 bits);
- **Traffic Class** (8 bits);
- **Flow Label** (20 bits).

What can the optimal order of these fields look like? In modern computer systems, proper alignment facilitates fast data access, so let’s start with the most essential fields, that is, the addresses. Ideally, they could do with a 128-bit alignment but the only way to provide for it without extra padding would be putting them in front, where the tiny **Version** field really needs to be for IPv6 to comply with the generic IP header format—see Chapter 1. So we have to lower our demands to the next power-of-two boundary, that is, 64 bits or 8 bytes. This automatically places the addresses behind all other fields because there are no other 8-byte boundaries available to them in a 40-byte header where offset 0 is already taken. Next, **Payload Length** can

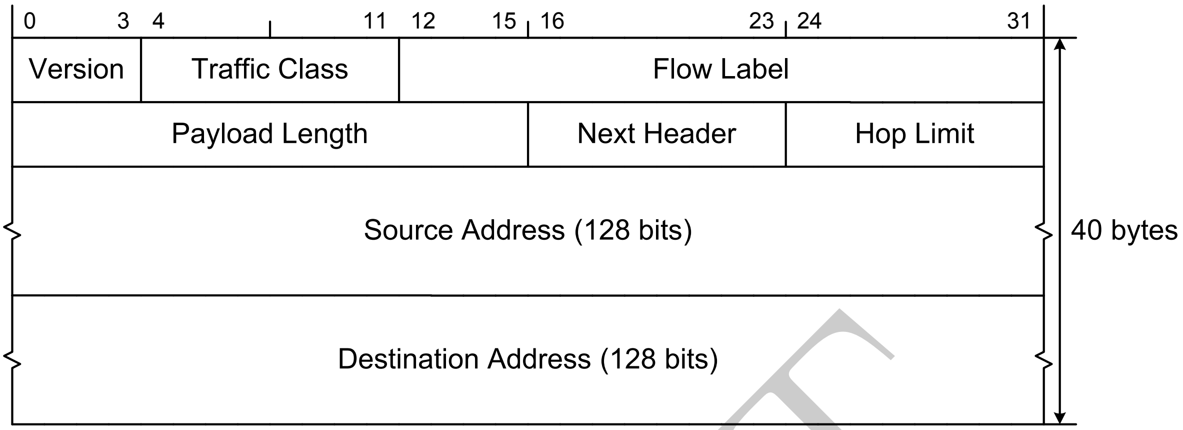


Figure 3.6: The IPv6 header

benefit from a 16-bit alignment, which can be readily provided by placing the QoS-related fields between **Version** and **Payload Length**. Finally, **Next Header** and **Hop Limit** find their place between **Payload Length** and the addresses. That's it about the IPv6 header format [82, Section 3]; now we can provide its graphical illustration—see Fig. 3.6.

Which fields we knew in the IPv4 header haven't made it into its IPv6 counterpart?

Firstly, **Header Length** was made redundant by the IPv6 header having a constant length of 40 bytes. The IPv4 header had a variable length and needed a Header Length field only inasmuch as IPv4 options were part of it, but IPv6 options will be isolated in dedicated extension headers, to be introduced in Section 3.3.2.

Secondly, all fragmentation related fields were left out: **ID**, **Offset**, and **Flags**. As long as the network is optimal and only a small fraction of packets get fragmented, these fields can reside in a dedicated extension header so that the non-fragmented majority of packets don't have to carry the overhead. The presence of such a header will be sufficient to flag a fragment. And what about the Don't Fragment flag (**DF**), significant only in non-fragmented packets and forbidding their fragmentation? This is a very valid question but it will have to wait until Section 3.3.4, where the destiny of the **DF** flag will be clarified.

Thirdly, **Header Checksum** was dropped. There are several reasons behind this decision. First of all, in the modular packet header structure of IPv6, the border between the header and payload becomes blurred. While the final destination can scan through all of the extension headers to locate where the upper-layer protocol data starts, a router cannot, and should not, look so deeply into each packet passing through. If so, a hop-by-hop Header Checksum can protect the integrity of the main IPv6 header only, unable to cover other IPv6-related headers and thus largely losing its point. Moreover, modern link-layer protocols such as *Ethernet* and PPP effectively take care of hop-by-hop integrity protection themselves through a consistent application of checksums and CRC codes while transport-layer protocols such as TCP and UDP utilize end-to-end checksums also covering a pseudo-header containing the IPv6 fields that don't change in transit, as we will discuss in Section 4.2. Hence no Header Checksum in the IPv6 header; in return, wire-speed packet forwarding will be facilitated and hardware circuit complexity, reduced as there is no checksum to verify and update in each packet forwarded.

> The reader will draw the IPv4 and IPv6 header formats side by side and fill their fields with different colors or shades based on whether the field was dropped, carried over, or born in the protocol upgrade. (It will make for a nice picture to hang on the wall afterwards.)

> Another exercise will be to work out which header fields will get corrupted if a router module fails to check the **Version** field and handles an IPv6 packet as though it were an IPv4 one. Consider the following cases: a) the packet is to be forwarded and so its **TTL** gets decremented; b) a QoS rule rewrites the **DSCP** value. (Hint: *Keep in mind that the* ***Header Checksum*** *will get updated as well.*) Next work out which of its header fields will get messed up if an IPv4 packet is processed assuming the IPv6 header format by mistake; consider just the case of **Hop Limit** getting decremented.

3.3 Extension Headers

3.3.1 A Header That Isn't There: No Next Header

Suppose an IPv6 packet consists of nothing but the main header. Although this packet's usefulness can seem a bit on the questionable side to us, there is no formal reason to prohibit such a packet either.

> In Section 3.3.5, we will find a use for such "empty" packets: traffic flow confidentiality.

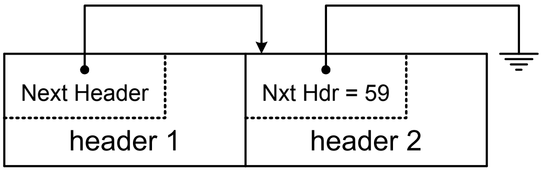


Figure 3.7: No Next Header

If a packet without payload has the right to exist, what does its **Next Header** value need to be? It could be argued that this value is irrelevant because **Payload Length** is zero anyway, but an ambiguity of this sort will be unwelcome in a protocol specification. In order to provide for consistent and predictable implementations, let's assign a single unique code from the IP Protocol registry [83] to this edge case. It will be 59, known as *No Next Header* [82, Section 4.7]. This code will be as useful in terminating a chain of multiple headers if there is nothing beyond it in the packet, as shown in Fig. 3.7.

And what if a packet's last header has **Next Header** of 59 but there are extra bytes after it as indicated by **Payload Length**? While there is no obvious reason to originate such a packet, it can still come from the network and we can't really say it breaks any laws or poses a threat to the nodes handling it provided they do the right thing. What the right thing is here can be figured out by Postel's Law: Intermediate nodes should preserve those trailing bytes unchanged and the final destination should ignore them [82, Section 4.7].

Let's recall the modern wording of **Postel's Law**, also known as the **Robustness Principle**. Originally, it was distilled by the Internet pioneer Jon Postel into this famous phrase: *be conservative in what you do, be liberal in what you accept from others* [94, Section 2.10]. Today we would add: *. . . but never trade security off.* This principle is a cornerstone of TCP/IP. If a protocol party sticks to it, that buys not only compliance with the specification but also flexibility and robustness to other parties' deviations and faults, and these qualities are paramount when the protocol is complex and its developers can't provide beforehand for every scenario or failure mode possible.

3.3.2 IPv6 Options

As we penciled it in back in Section 3.1, there will be two classes of IPv6 options, hop-by-hop and destination, their handling requirements significantly different. First, routers are to scan only through hop-by-hop options for the sake of performance. Second, when IP gets end-to-end security, destination options may be fit for encryption whereas hop-by-hop options will definitely not. Third, when it comes to fragmentation, all hop-by-hop options present in the original packet will need to be replicated in each of its fragments while destination options will need no such treatment: they can even be fragmented instead if they don't fit in one fragment.

> These are general considerations. The actual rules of header fragmentation will be worked out in Section 3.3.4, when we are ready for that.

Everything considered, these two option flavors deserve different extension headers just to keep them separate. The format of those headers can be just the same, but they will have different **Next Header** codes assigned to them. In particular, the Destination Options header will have a **Next Header** code of 60 [82, Section 4.6].

The Hop-by-Hop Options header is special in that it needs to be easily accessible by routers. For its role, it will be given a distinctive **Next Header** code of 0 and required to immediately follow the main IPv6 header. In other words, a **Next Header** value of 0 may appear only in the main IPv6 header. If a packet's very first **Next Header** value is non-zero, then there are no hop-by-hop options in the packet—assuming the packet is valid, of course. This simple logic will allow routers to locate hop-by-hop options with minimal effort.

To optimize option encapsulation, the options headers will be containers capable of including multiple options each. If individual options utilize TLV coding, the fixed part of the options header format can be limited to just two fields: **Next Header**, and a length to provide for a variable number of options accommodated.

> If we really wanted to, we could do without a length field through a special way of encoding option data. However, that would be an unnecessary complication of the matter whereas an explicit length value stored at a fixed offset provides for locating the header end easily. Moreover, explicit length information will allow to skip over the header without having to decode its contents.

Next Header is a one-byte field. Let the length field occupy another byte so that the actual option data is aligned on a 16-bit word boundary. If so, the length field can only count up to 255, which may not be quite enough

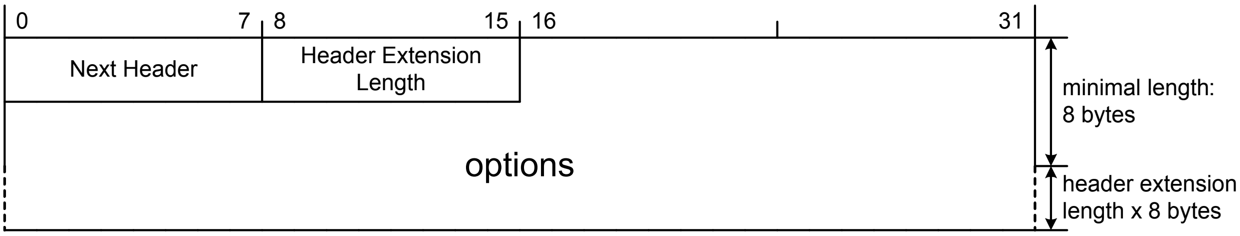


Figure 3.8: Options header common format

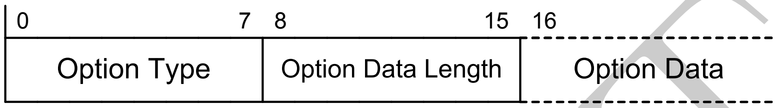


Figure 3.9: IPv6 option format

for a container with multiple options in it. This concern can be addressed by using a length unit larger than the byte, also providing for a nice alignment of the header end. The end of the main IPv6 header is aligned on an 8-byte boundary and so it will be pointless to try to achieve a greater alignment in the extension headers to follow. Based on this, the options header length will be stored in 8-byte units. The smallest total length possible will be 1 unit because of the fixed fields, so the 1 can be taken away from the final value stored so as to gain 8 more bytes in the available length range. Thus the length field stores the extra length, counting from the second unit of 8 bytes. Hence the field name: **Header Extension Length**, abbreviated to **Hdr Ext Len** in some RFC documents. Due to this format, shown in Fig. 3.8, the total options header length can vary between 8 and 2048 bytes in 8-byte increments.

Individual options will be TLV-encoded as shown in Fig. 3.9: a type field followed by a length field followed by option-specific data. The type and length fields will occupy 1 byte each. The option length will be in bytes and, as one would expect in IPv6, won't include the fixed fields of type and length: it is only option data length that is stored.

While scanning through the IPv6 options in a packet just received, a node might stumble upon an unknown option corresponding to a protocol extension unsupported by the local IPv6 stack. Thanks to the uniform TLV encoding, the unknown option is easy to skip over; but is it really OK to ignore that option? To avoid any doubt, let's have the option's handling requirements encoded in the high bits of its type value as shown in Table 3.1.

Table 3.1: The meaning of the high bits in an IPv6 option type

Bits	Meaning
00	It is OK to ignore this option.
01	If unsupported, the packet is to be silently discarded.
10	If unsupported, the packet is to be discarded. An ICMP-like notification is to be sent back to its source regardless of the destination address type.
11	If unsupported, the packet is to be discarded. An ICMP-like notification is to be sent back to its source unless the destination address was multicast.

In Table 3.1, we assumed that IPv6 would eventually be supplemented with a control protocol comparable to ICMP in IPv4. Real work on it won't start until Section 4.3, but its name can already be guessed: ICMPv6.

Yet another issue is going to crop up when there is end-to-end security available in IPv6. Can options be included in the scope of integrity assurance then? Obviously, there will be no simple answer to fit all options because some options can be altered en route while others should never be modified once the packet was originated. On the one hand, protecting a volatile option will result in the packet discarded by its destination as corrupted; on the other hand, non-volatile options can be protected all right. The best we can do here is have this property encoded, too, in the third most significant bit of the option type value, e.g., as follows: 0 means an option whose data is known to stay the same end-to-end and 1 means that the option data may change en route. Then the security mechanism won't need to know the properties of all possible option types beforehand—which would be just impossible—because it can simply test the 3$^{\text{rd}}$ bit of the type value.

For example, if the "Can Change En Route" bit is set, IPsec AH [95] will use zero-value bytes instead of the actual option data bytes, byte for byte, for the purpose of computing or verifying the packet's integrity check value. Thus, at least the option type and length fields will be protected along with the total option length, assuming an unchanging option length and provided the integrity check algorithm is sensitive to the number of zero bytes coming in a run. More on this in Section 3.3.5.

The three high bits of an option type value have the option's inherent properties encoded in them as shown in Fig. 3.10, and those properties are

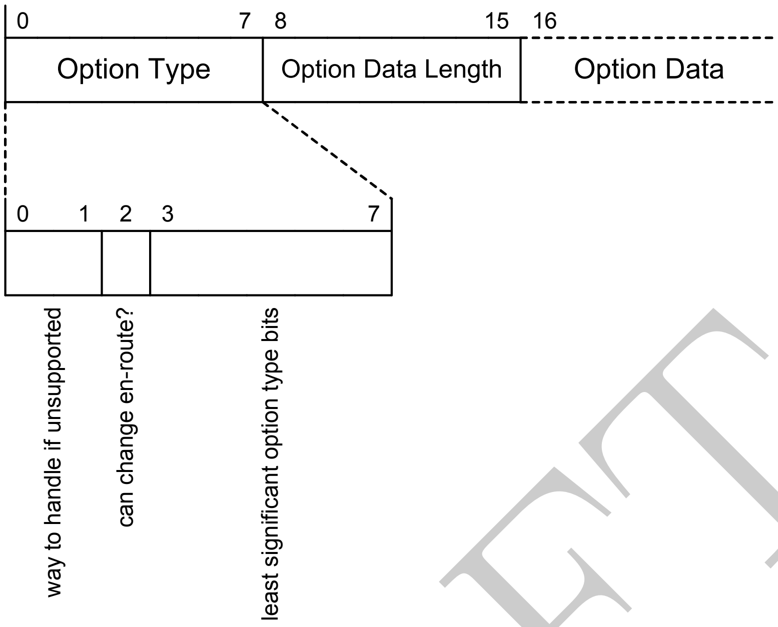


Figure 3.10: Encoding option properties in its type

the same for all options of this type. By design, the high bits of an option type value are its integral part rather than a separate protocol field and they can't be tweaked in option instances. Instead, they are set accordingly when the option type is created and never altered after that. For example, if we encounter an option of type 95 decimal, we will be immediately[1] able to tell the following facts about this option:

- This option can be modified en route.
- If a node doesn't support this option, it needs to silently drop the whole packet.

And if, say, the "Can Change En Route" bit is flipped in this type value, the result will be a completely different option type having no connection with the original one.

Calculate the decimal value of the resulting option type. (Hint: *127*)

The last thing to say about IPv6 option types is that, although hop-by-hop and destination options are kept separate in different extension headers,

[1] Or as soon as we manage to convert 95 to binary.

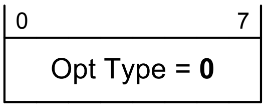


Figure 3.11: *Pad1* option

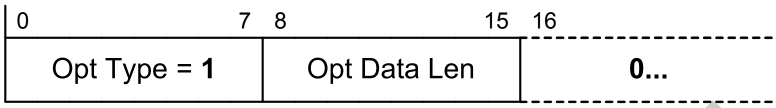


Figure 3.12: *PadN* option

they share one option type registry [96] and so one type value can never mean two different options.

The overall length of options to be put in a packet can be any integral number of bytes. At the same time, the options header length needs to be a multiple of 8 bytes, leaving $8N - 2$ bytes for the options proper. Apart from that, some options may benefit from a certain alignment within the packet. In order to meet these requirements, let's introduce two dummy options whose support will be mandatory—oxymoron!

The first of them, to be able to pad things up by just 1 byte, has to be 1 byte long whereas the shortest conformant IPv6 option possible will be 2 bytes long. Therefore an exception has to be made for this option, to be known as *Pad1*: It will consist of just 1 byte with a value of zero as shown in Fig. 3.11. Although its nominal type is 0, this IPv6 option doesn't really adhere to the TLV-based format and fortunately is the only one not to do so.

Then the other dummy option, *PadN*, is able to comply with the IPv6 option format as shown in Fig. 3.12 because it will take care of padding from 2 through 257 bytes. Its well-known type value is 1. When its data length is greater than zero, its data field is to be zero-filled in keeping with the well-known protocol design rule that dummy data be zeros.

By their role, *Pad1* and *PadN* options are free to appear in an options header once or multiple times and to come at its beginning or tail, or between other options. Moreover, they belong in both a Hop-by-Hop Options Header and a Destination Options Header.

Analyze the properties of the *Pad1* and *PadN* options as encoded in the high bits of their type values.

All other IPv6 options are protocol extensions specified in separate RFC documents. We won't discuss them now, hoping that by the end of the book the reader will have enough background to make sense of them when needed.

> We are still going to "create" one meaningful IPv6 option, known as Router Alert Option, in Section 6.4.

3.3.3 Routing Header

An important corollary of the end-to-end principle dominating the Internet is that hosts aren't really involved in IP packet routing. An IP packet's source only creates the packet and injects it into the network for the latter to work out a suitable path to the packet's destination. A network engineer certainly knows that the path from the source to the destination can span multiple links and routers operating to certain protocols and specs, but those gory details are of no concern to the packet's source: All it needs to do is create the packet and pick its first hop. By this model, the source can influence the subsequent routing of the packet only indirectly, by setting its Destination Address to a certain value.

> To make a routing decision, a modern IP router implementation can consider other packet fields and characteristics as well, such as the traffic class or the total length. It is a generalization of the classic routing algorithm, whose output (the next hop) depended only on the Destination Address. Routing so generalized is known as **policy routing**. Nevertheless, even with policy routing in use the packet's source can control the packet's path only indirectly.

Once, in a momentary departure from the end-to-end principle, it occurred to the creators of IPv4 that a situation could come about where the source was better informed than the network. Suppose for example that the routing protocol has distributed false information due to an attack or fault and now the routers are forwarding packets down the wrong path. In this case, the source could tell the routers the right path by including the relevant information in the IP header. That was how IPv4 got its source routing options: LSRR and SSRR [85, Section 3.1].

> There is an interesting draft document on IP source routing option design by the founding fathers of the Internet J. Postel and J. Reynolds [98]. It was never finalized but it still provides insight into the decisions made back then. In particular, it can be learned that there were two goals for IP source routing to achieve. One of them was to direct the packet along a certain path disregarding the current state of the intermediate nodes. In other words, specific nodes were to be included in the packet's path in certain order. The other goal was to exclude undesirable nodes from the path by having the packet bypass them. The technical solution was just the same in both cases and consisted in listing the nodes for the packet to travel through. It was made so because it wasn't practical to instruct an intermediate IP node such as a router *to not forward* the packet via a specific next hop X, for it could have no other option but X due to its routing algorithm. For this reason, bypassing unwanted nodes had to be implemented by means of including alternative ones in the packet's path.

As we may still remember, the principle of operation of the xSRR options was pretty simple in IPv4. Suppose source S needed to ensure that the packet to final destination D would travel via intermediate nodes T_1, T_2, T_3,... T_N. In that case, the packet header has to be filled in as follows:

- The address of T_1 was put in the **Destination Address** field of the IPv4 header.
- The list of addresses $\{T_2, T_3, \ldots T_N, D\}$ was stored in the xSRR option.
- The pointer in the xSRR option was set at the first address in the list, that is, T_2.

So node T_1 initially appeared as the packet's destination and was the first to receive the packet. It would alter the packet header as follows:

- work out which address the xSRR option pointer was set at: T_2;
- copy address T_2 over to **Destination Address**;
- work out the egress interface using the routing algorithm on the new **Destination Address**;
- store the local address of the egress interface to where T_2 was in the xSRR option address list;
- advance the xSRR option pointer to the next address in the list, i.e., T_3.

Next, the packet would reach node T_2 and get processed the same way, thus hopping on to T_3, and so on until it ended up at its final destination D. The difference between the loose mode and the strict mode of IPv4 source routing as represented by the LSRR and SSRR options, respectively, was that in loose mode the packet could visit other intermediate nodes in addition to what were specified in the list and in strict mode it couldn't: All of the nodes, S through T_1 through T_N through D, needed to be chained with direct links for the packet to successfully arrive at D when in strict mode.

In hindsight, we can't help wondering why the network was to trust its end hosts that much. Indeed, a malicious source can make use of the source routing options for at least two evil purposes:

- Security circumvention. Suppose a network has its routing set up so that all its traffic passes through a firewall, where it is subject to security checks. Thanks to source routing, an attacker can bypass the firewall via other nodes if the physical topology of the network leaves such a loophole.

- DoS attack amplification [99]. Suppose an attacker wants to suppress an underpowered node A with a flood of IPv4 packets. If an LSRR option is included in each packet with the following source route list: $\{B, A, B, A, B, A\ldots\}$, where B is another node reachable from A, then each packet will visit A not just once but multiple times because it will be ping-ponged between A and B until the source route list is exhausted. Consequently, a packet flood N times as weak will be sufficient to mount an attack of the same intensity as without the use of LSRR, and such a flood will have a better chance of getting through the network perimeter unnoticed.

The reader will calculate the maximum DoS amplification ratio N_{max} achievable this way in IPv4.

Solution: The longest IPv4 header possible is 60 bytes because the length unit is 4 bytes and the length field is 4-bit and so can count up to 15. Of the 60 bytes, 20 bytes are occupied by the fixed part of the IPv4 header, leaving up to 40 bytes to IPv4 options. The fixed part of the LSSR option is 3 bytes but the smallest valid offset value is 4 bytes [85, Section 3.1] so the destination list can span up to 36 bytes, that is, include up to 9 IPv4 addresses. An additional destination address is in the fixed IPv4 header, making 10 destination addresses in total. Therefore each of the alternating addresses A and B can appear in the list 5 times at most. This will be the greatest DoS amplification ratio achievable in IPv4.

All in all source routing in a public global network can do more harm than good, and it would be unwise to repeat the past mistakes of IPv4 now, in IPv6. To avoid that, we will play smart and provide just a most general format for the extension header to control IPv6 packet routing but leave all details of its actual use to supplementary protocols. So we won't have to take the blame if the use of that extension header results in security issues.

The extension header in question will be known as the **Routing Header** [82, Section 4.4]. It is identified by a **Next Header** value of 43—of course, to be found in the preceding header, not in the Routing Header itself. (See Fig. 3.3 in page 94 to refresh your memory on how headers are linked in an IPv6 packet.)

Source routing information will be of variable length, and so will the Routing Header. Therefore there will need to be a length field in it in addition to the mandatory **Next Header** field. Along the lines of how the options headers were laid out in Section 3.3.2, the length field will be 1-byte and count 8-byte words less the first word of the header, which is always present due to the fixed fields. Hence a familiar name: **Header Extension Length**.

Following the essential fields of **Next Header** and **Header Extension Length** will be a **Routing Type** field whose value will tell how this Routing Header instance is to be interpreted. One byte will be quite enough for it, and a registry of its well-known values will certainly be required [96].

What can the receiver of a packet with a Routing Header do if the **Routing Type** value is unknown or unsupported? Will such a packet always have to be discarded? We can answer this question based on our experience with IPv4 source routing—one can learn from mistakes, too! When the source route list in an xSRR option had been exhausted and the packet was heading toward its final destination D, the xSRR option's mission was complete and its information was no longer relevant for routing the packet. In that case, the xSRR option contents could be just ignored.

To reuse this idea in IPv6, a certain structure shared by all Routing Types has to be imposed on the source route data. Let it consist of segments, where a segment can be an IPv6 address or whatever, depending on the **Routing Type** value. How can it be indicated that all of the segments have been handled and subsequent nodes in the packet's path can just ignore its Routing Header regardless of its **Routing Type** value? Here is a possible solution: to store the number of segments yet to be handled in the fixed part of the header. As soon as this number drops to zero, it will be evident that the Routing Header has completed its mission and can be ignored altogether; but until that it won't be safe to disregard the header if it is unsupported and the whole packet will have to be dropped. The 1-byte field to store the number of active routing data segments is known as **Segments Left**.

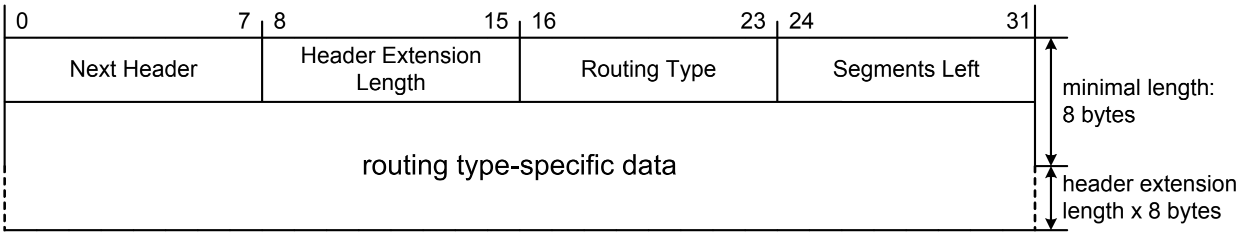


Figure 3.13: Routing Header common format

The resulting format of a generic Routing Header is shown in Fig. 3.13.

Of the Routing Header types defined, we are going to mention **RH0** (Routing Header Type 0) and **RH2** (Routing Header Type 2). RH0 is to IPv6 just what LSRR was to IPv4. It is no surprise that RH0 has eventually been deprecated for the sake of security [100]. RH2 will be explained to the curious in a remark at the end of this subsection.

One difference from LSRR that RH0 has is as follows. The current segment of RH0 is overwritten with the former IPv6 Destination Address instead of the egress interface address. Thus processing RH0 can be as simple as swapping the current segment address and the Destination Address followed by decrementing the **Segments Left** counter. The end result in RH0 will be just a circular shift of the source-route list whereas LSRR would record the actual interfaces used to forward the packet. Those interface-recording semantics were, in fact, required by its sibling SSRR so that the reply packet could include a usable strict source-route list too.

As long as a private backbone network has a proper border security policy blocking untrusted RH0 from the outside, inside such a network RH0 could be a valuable tool for the discovery of latent problems. Through the use of RH0 a single monitoring station could be probing different paths through the network including the standby ones. Correlating the results from such probes over an exhaustive set of paths can be an effective method to locate hidden issues such as faulty backup links with a great precision [101]. So, admittedly, the deprecation of RH0 was prompted not by its uselessness or inherent flaws but mostly by the insufficient network security culture still dominating the Internet and making this tool dangerous for its operator.

That a Routing Header exists in IPv6 makes us reconsider the notion of packet destination. Until now, our assumption has been that a packet has just one destination, be it a single host or a multicast group. But from now on two destination types will have to be distinguished. If there is no Routing Header in a packet at all, or if there is one but its **Segments Left** value is zero, the **Destination Address** from the main IPv6 header is that of the packet's **final destination**. The latter is the node to ultimately consume the packet and process its upper-protocol payload. On the other hand, should there be any active segments still left in the Routing Header, as indicated by a non-zero **Segments Left** value, the main IPv6 header will contain the address of the **current destination** of the packet. That node will receive the packet only to change its **Destination Address** and pass it on; so it will be an intermediate node in the packet's path. In that case the final destination address is stashed in a Routing Header segment.

Generally, none of the destination addresses of a packet having a Routing Header in it may be multicast. This follows from basic considerations of network security.

What rules will be needed to handle non-global destination addresses in a Routing Header? The zone isolation principle from Section 2.4 will be sufficient for that. But, strictly speaking, that principle will need to be met by, and enforced for, all destination addresses in the Routing Header at once; otherwise it will be possible to exploit the Routing Header for inter-zone attacks. For example, if no such enforcement is done, a packet whose current destination is global and final destination is link-local can end up injected into a remote link-local zone having no direct connection to the packet's source node.

There is an RFC requirement [46, Section 9] that the next destination address from the Routing Header have scope not smaller than that of the current Destination Address in the main IPv6 header. As long as the zone isolation principle from Section 2.4 is met and all of the packet's destination interfaces are within its Source Address zone, that requirement can be excessive. For example, a packet's destination addresses can be of tapering scope if all of them belong to interfaces into just the same link the source interface is on.

Source routing has effectively been revived in MPLS-TE, where it is perfectly possible to specify which nodes an LSP needs to go through by listing their addresses in an ERO. Of course, this mechanism of MPLS is superior to the IPv6 RH0 as far as security is concerned because the edge router (LER) signaling the LSP is fully trusted, unlike an arbitrary IPv6 source out there on the 'Net. However, MPLS isn't a completely independent stack: Having no addressing of its own, MPLS has to rely on a conventional Layer 3 protocol such as IP in this respect, and the zone isolation principle will inevitably apply to the ERO if the MPLS nodes involved happen to have scoped IPv6 addresses. That is, each address listed in the ERO will have to be reachable from the ingress LER signaling the LSP; otherwise the ERO will be meaningless. In particular, this is why an LSP spanning multiple links can't be signaled using only link-local addresses: Addresses of larger scope will be needed in the ERO, at least from the second one on. It is really nice to know that our work on RH0 has not been pointless because some of our findings are as applicable to the mighty MPLS!

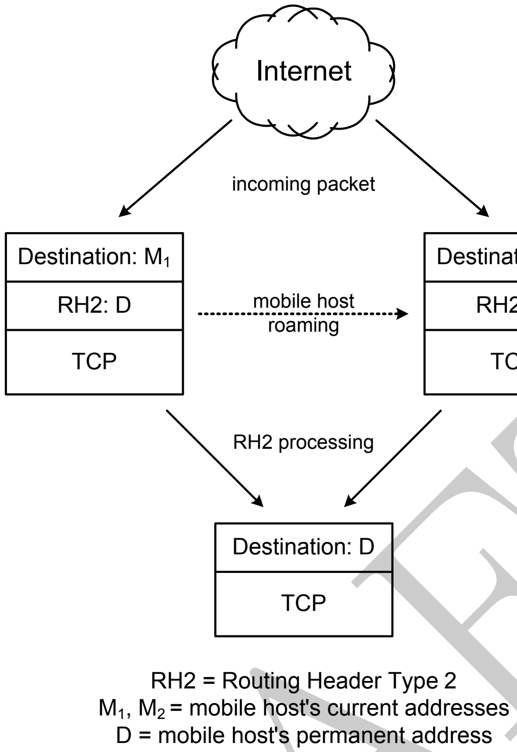


Figure 3.14: How RH2 works

As we know, an IP host can address and send packets to itself. In a similar vein, the current and final destination addresses of an IPv6 packet can belong to the same host. What can that be good for? For one thing, the current destination address can be the host's locator and the final destination address, its identifier. This can be useful when the host is mobile and roams around the Internet. In that case, the mobile host's current address M will be reachable through regular routing but it will have to change each time the host moves onto a different link. At the same time, the mobile host's permanent address D visible to the upper-layer protocols will be reachable only thanks to a Routing Header. The host thus will be able to roam in a manner transparent to any protocol above IP as illustrated in Fig. 3.14. Quite clearly, such a Routing Header may contain just one segment and the final destination address conveyed so must be equal to the mobile host's permanent address D. Otherwise it is an evident protocol fault—or a malicious attack. This type of Routing Header has been assigned Type 2 and so it is known as **RH2**. It is defined in [102], Mobility Support in IPv6.

3.3.4 Fragment Header. IPv6 Fragmentation and Reassembly

As we should remember, IPv4 fragmentation was a mandatory function in both hosts and routers because an IPv4 packet exceeding the link's MTU had to be fragmented on the spot by the forwarding node. Needless to say, this contributed to the complexity of IPv4 router implementations since they couldn't just regard a packet as a data unit to be forwarded at once or dropped. In addition, concerning routers with fragmentation was a less than optimal choice because a router could be handling traffic for many hosts whereas a host cared only for itself as far as the IP layer went.

Now we have a nice opportunity to turn the protocol on its head, so let's try fitting these new rules of fragmentation onto the IPv6 framework:

1. A packet can be fragmented only at its source.

2. Intermediate nodes including routers and non-final destinations will forward packets with no regard to whether those are fragments or whole packets.

3. A fragmented packet is reassembled at its *final destination*, as before.

Review Section 3.3.3 for the distinction between the final and non-final (current) destinations of an IPv6 packet.

Thus IPv6 fragmentation is going to be solely the packet source's job, which is both fair and advantageous to the performance of the network.

Unfortunately, there seems to be no explicit specification in [82] that it is the final destination that reassembles a fragmented IPv6 packet, although this can be inferred from the recommended order of extension headers [82, Section 4.1]—to be discussed in Section 3.3.6.

The new approach to fragmentation is likely to pose new problems, but they are just the price of progress and we will have our share of fun solving them.

As soon as packet length L becomes an end-to-end property, to never change en route from the source to the final destination, the packet will be able to reach its final destination only provided the following condition is satisfied:

$$L \leq PMTU. \tag{3.1}$$

Here $PMTU$ is the Path MTU value of the path from the source to the final destination for the packet to take:

$$PMTU = \min_i MTU_i, \tag{3.2}$$

where i indexes all egress interfaces in the packet's path.

In the new order of things, L will be controlled by the packet source, so it is up to the latter to make sure the condition from Equation 3.1 is met. This depends on working out the value of $PMTU$ or at least a lower bound to it, $\underline{\mathrm{PMTU}}$:

$$L \leq \underline{\mathrm{PMTU}} \leq PMTU \Rightarrow L \leq PMTU. \tag{3.3}$$

By trivial math, there are two strategies to satisfy Equation 3.3. Firstly, the source can put some effort into finding a good lower approximation to $PMTU$, up to its exact value. That's what the PMTU Discovery procedure, aka PMTUD, is for. For it to work, the source needs feedback whenever its packet is discarded due to exceeding MTU. IPv4 had ICMP Packet Too Big messages for that purpose; a similar device will be required in IPv6. PMTUD should complete faster if the actual MTU_i value that blocked the packet is signaled back to the source. However, in IPv4 this was an extension to the original Packet Too Big message format not all nodes supported: The MTU_i value was stored in a formerly unused field [103, Section 4]. Now such a signaling will need to be provided for in the base protocol—something for us to keep in mind until Section 4.3. Next, PMTUD assumes that the upper-layer protocol can cope with packet loss or doesn't care about it. This condition is nothing new as packets get dropped in TCP/IP networks for a variety of reasons apart from exceeding MTU. Finally, PMTUD can be reliable only providing that either at least a fraction of the Packet Too Big notifications reach the source OK or the upper-layer protocol involves reply packets from the far end. If neither condition is met, it is known colloquially as a PMTUD black hole, where large packets vanish unbeknownst to the source.

A classic PMTUD black hole appears if Packet Too Big messages are blocked by a misconfigured firewall. Not all is lost then for a bidirectional protocol such as TCP though: No response from the far end to a large packet, even if retransmitted a few times, can be regarded as a symptom of a PMTUD black hole in the path. Then the source can try to work around the black hole by reducing the packet size although Packet Too Big messages never reach it. By contrast, a unidirectional protocol such as Syslog or Netflow can provide no such feedback, or lack thereof, to the source, and its packets will be disappearing in the black hole until Packet Too Big signaling is set right by unblocking the relevant ICMP type and code in the firewall or getting the firewall software relate ICMP error messages with their respective flows.

The other strategy to tackle $PMTU$ approximation is suitable for sources that can't be bothered to carry out PMTUD, e.g., due to having very limited computing resources. Namely, a source can be using the minimal MTU value as specified at the IP level to ensure PMTU clearance of the packets originated:

$$L \leq MTU_{min}. \tag{3.4}$$

This trick can work because the minimal MTU value is guaranteed to be a lower bound of PMTU for any path compliant with the IP specification:

$$\forall i, MTU_{min} \leq MTU_i \Rightarrow MTU_{min} \leq PMTU. \tag{3.5}$$

To avoid a heavy performance penalty when using the second strategy, MTU_{min} needs to be reasonably large in IPv6 and, for one thing, considerably larger than the tiny MTU_{min} of 68 bytes adopted by IPv4. To pin the IPv6 MTU_{min} value down, let's prescribe that no IPv6-ready link shall ever have MTU less than 1280 bytes [82, Section 5].

Honestly, we have no idea why 1280 bytes. Of its obvious arithmetical properties, this one can be noted: $1280 = 2^{10} + 2^8$. However, what has greater practical importance is that this value leaves enough headroom for multi-level encapsulation, e.g., that employed by various kinds of network tunnels, as long as the underlying "physical" MTU stays around the popular value of 1500 bytes.

But what if a link technology can't easily provide for an MTU as large as 1280 bytes, e.g., due to its physics? (For a real-world example, refer to [104].) In this case, multi-layer network architecture has to come into play yet again: Extending the effective MTU to 1280 bytes or more by means of frame fragmentation and reassembly will be the job of an adaptation sub-layer to be placed on top of the raw link layer and below IP, usually a part of the corresponding link-layer encapsulation protocol. (See [105] for a real-world example.)

Back in Section 3.2, we wondered where the **DF** flag had gone from the IPv6 header. Now we have naturally come to the conclusion that it is no more needed in IPv6. In a sense, the **DF** flag is implicitly set in every IPv6 packet because no intermediate node may fragment it.

As it was already decided in Section 3.2, the other fragmentation-related fields need to go into a dedicated extension header. It will be known as the **Fragment Header** and have the IP Protocol / **Next Header** value of 44 assigned to it. Its presence will flag IPv6 fragmentation: If a packet has a Fragment Header in it, it is a fragment; otherwise it is a whole packet.

Since a fragmented packet can be reassembled only by its final destination, each fragment has to carry enough information as required by the intermediate nodes. That is, each fragment needs to be a valid IPv6 packet in its own right, able to travel through the network independently of the other fragments.

First and foremost, the main IPv6 header needs to be present because it contains the most basic info about the packet. Next, any hop-by-hop options are of relevance to all nodes in the fragment's path. A Routing Header, if present in the original packet, will be relevant to the intermediate destinations albeit disregarded by routers. Finally, the destination options *might* be of interest to intermediate destinations. It is quite likely that there will be two kinds of destination options: those meaningful to all destinations and those significant to the final destination only. If so, the former options can be isolated in a Destination Options Header to be included in each fragment while the latter options can be freely fragmented along with the remaining extension headers and payload.

Note well that the final destination is concerned with *all* extension headers.

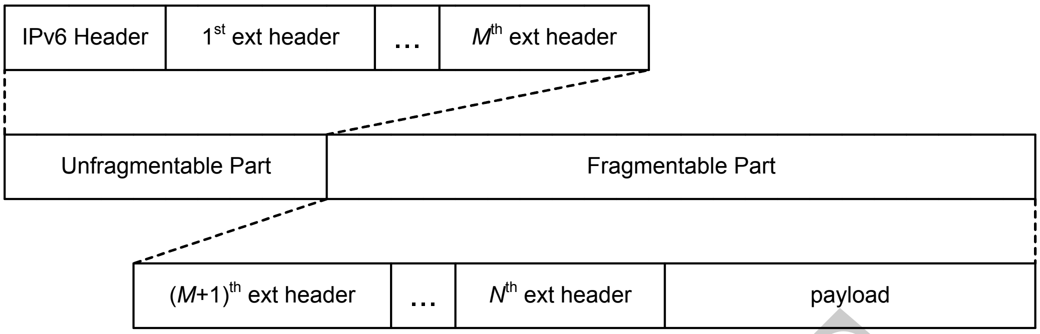


Figure 3.15: IPv6 packet layout for fragmentation purposes

> The options to be seen by all destinations and those for the eyes of the final destination only will use just the same extension header type, Destination Options (60). The difference will only be in their relative placement in the packet, the former preceding the Fragment Header and the latter coming after it.

Generalizations being our soft spot, let's outline the IPv6 fragmentation procedure first and only then proceed to define its details. The input to the fragmentation procedure is a whole, unfragmented, IPv6 packet. Our theory goes that some of its headers need to appear in each fragment while the other headers and the payload can be split between the fragments. So it can be said that for fragmentation purposes the original IPv6 packet consists of two parts: **Unfragmentable** and **Fragmentable**. As we have just figured out, the Unfragmentable Part consists of the following headers:

- the main IPv6 header (always);
- the Hop-by-Hop Options Header (if any);
- the Destination Options Header for intermediate destinations' eyes (if there are such options in the original packet);
- the Routing Header (if present).

The rest of the extension headers together with the payload constitute the Fragmentable Part of the packet. This layout is summarized in Fig. 3.15.

> Suppose an IPv6 packet is transmitted through the network fragmented. Where and in what form do you think the original packet ever exists? (Hint: *In the memory buffers of its source and final destination.*)

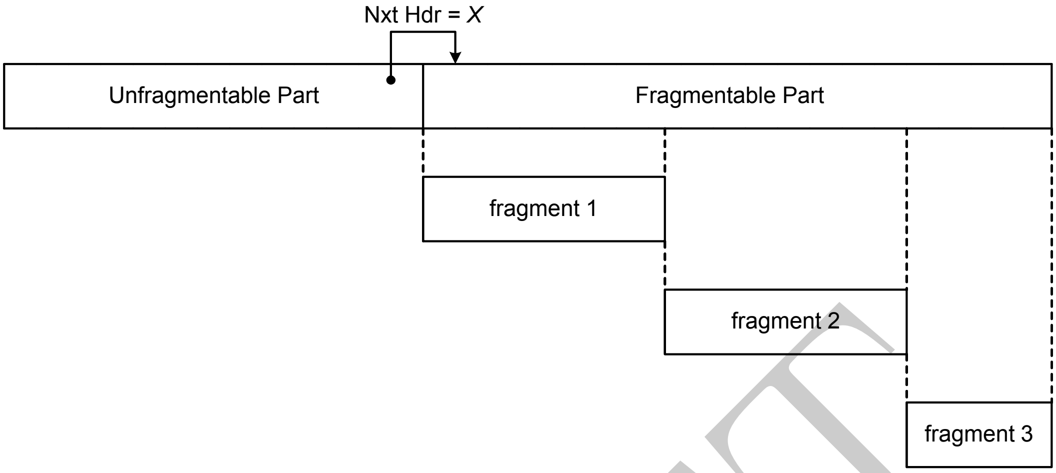


Figure 3.16: Making raw fragments from an IPv6 packet

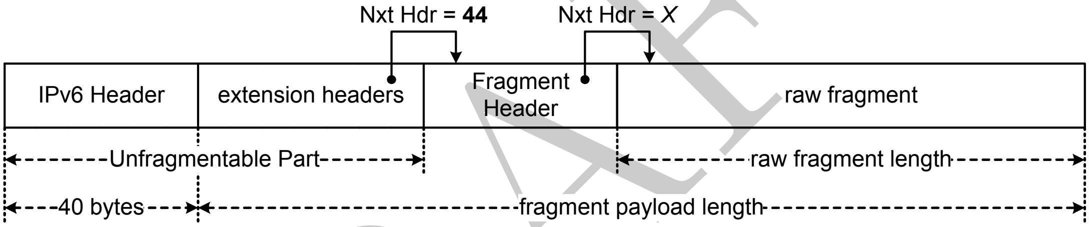


Figure 3.17: Encapsulated IPv6 fragment

First thing in the fragmentation procedure, the Fragmentable Part is chopped up into raw fragments of non-zero length so that no fragment's length plus the encapsulation overhead exceeds the current PMTU estimate. This process is illustrated in Fig. 3.16. What the encapsulation overhead is going to be we will see in the next step of the encapsulation procedure.

In the next step, each raw fragment is turned into a valid IPv6 packet. This is done by prepending a Fragment Header to it and then prepending a complete copy of the Unfragmentable Part before the Fragment Header as shown in Fig. 3.17. For the resulting IPv6 packet to become valid, just a couple of its fields need to be adjusted, namely, **Payload Length** in the main IPv6 header and **Next Header** in the header immediately preceding the Fragment Header. The first adjustment is necessary because **Payload Length** only pertains to this packet even if it is a fragment. The second adjustment is to allow for the change in the chain of headers: Now the Unfragmentable Part is followed by a Fragment Header instead of whatever was at the beginning of the Fragmentable Part.

> There are Layer 3 protocols in stacks other than TCP/IP whose headers have separate fields for this fragment's length and the original packet's total length. One example is ISO CLNP.

As we can see now, the encapsulation overhead on each fragment is due to the Unfragmentable Part and a Fragment Header. It needs to be allowed for when computing the maximum raw fragment length from the current PMTU estimate: The former is equal to the latter less the lengths of the Unfragmentable Part and the Fragment Header. Will this computation be possible prior to actually fragmenting the packet so that the maximum raw fragment length is known at the right time and there is no chicken-and-egg problem? The answer is affirmative. Indeed, the Unfragmentable Part length only depends on the original packet structure and never changes from fragment to fragment. As for the Fragment Header length, we can just make it fixed so as to simplify the computation.

> A lawful source is not expected to vary the contents of the Unfragmentable Part between fragments of the same packet because there hardly is a good reason to do that. Still, a final destination can encounter a scenario where different fragments of one incoming packet have different Unfragmentable Parts. In that case, precedence goes to the Unfragmentable Part from the fragment having the offset of zero.

Now it is time we came up with a Fragment Header format, but it depends on the way IPv6 fragmentation and reassembly are going to be parameterized. We can avoid reinventing the wheel here and simply reuse the parameter set from IPv4 since it meets the most critical requirement of being able to cope with fragments delivered out of order. So the Fragment Header will need to have at least the following fields in it:

- **Identification**;
- **Fragment Offset**;
- **More Fragments** flag.

These fields need to be supplemented with **Next Header** to provide for IPv6 header and payload linkage. The resulting number of fields is fixed, and so can be the Fragment Header length, which is what we aimed for. Now let's carefully consider each parameter and, if necessary, adjust its specification for IPv6.

Identification was a 16-bit field in IPv4 and that proved insufficient as the traffic rate grew: At a fragment rate sufficiently high, mis-association of fragments was probable whereby unrelated fragments bearing the same ID value due to wrap-around would be combined into corrupted packets [106]. If fragmented traffic was pumped at an even higher rate, the IP data corruption rate would become high enough for the TCP and UDP checksums to start failing: They would no longer detect all corrupted packets because of checksum collisions when a corrupted packet had the right checksum. Therefore it is clear that IPv6 needs a longer **Identification** for its fragments. Will 32 bits be enough?

To work this out, we need first to define the scope where **Identification** needs to be unique. To give us a clue, the IPv4 ID wasn't supposed to be unique per se: Instead, it was combined with the Source Address, the Destination Address, and the Protocol value into a tuple, and it was the tuple that needed to be unique. Reusing the same tuple in IPv6 is problematic because there is no single Protocol in an IPv6 packet having multiple extension headers in it. For this reason, Protocol is excluded from the IPv6 fragment identification tuple and IPv6 Identification needs to be unique only for a specific Source Address and Destination Address pair.

If a packet has a Routing Header in it, it is the final destination address that goes into the tuple for fragment identification purposes [82, Section 4.5] since IPv6 reassembly is the final destination's job.

If the Source or Destination Address of a fragment is scoped, its zone index has to be considered in fragment identification [97]. Should it be overlooked, bad things can happen; namely, fragments coming from different zones of the same scope can be mis-associated if they happen to have the same ID and their numeric addresses match. For example, fragments with a Source Address of *FE80::1*, a Destination Address of *FE80::2*, and an ID of 42 should belong to different packets if they arrived on different links. This is a corollary from the IPv6 scoped address architecture we discussed in Section 2.4.

As for the time domain, Identification needs to remain unique only as long as any fragments of the packet live. A fragment's maximum lifespan can be estimated as the sum of the network-induced delay and the reassembly timeout. Unfortunately, the former component isn't really bounded and can be however large, at least in theory.

It would have been possible to accurately estimate a fragment's lifetime if IP TTL had still meant time in seconds. However, we had to move away from the wall-clock based interpretation of packet TTL for practical reasons in Section 3.2—see also [82, Section 8.2] on the issue of IPv6 packet maximum lifetime.

Therefore we will have to make the reasonable assumption that the network-induced delay effectively becomes smaller as the traffic rate goes up. The point is that a large delay has mostly to do with packet queueing in the intermediate nodes rather than with the physical signal propagation velocity. Ultimately, at gigabit rates, a packet will be dropped much sooner than held in the queue for a full second just because queuing a second of traffic would require a gigabit of buffer space.

The conclusion is that at a high traffic rate—the case of interest to us here—the reliability of reassembly will be mainly controlled by the reassembly timeout. This timeout needs to be shorter than the Identification wrap-around period. In its turn, the wrap-around period shrinks as the traffic rate grows, so its lower bound can be obtained from the upper bound of the traffic rate a single source can be transmitting to a single destination at.

The worst case will be that wherein Identification values are spent at the highest rate possible for a fixed bit rate. One Identification value is spent per original packet, so what we need to work out is the maximum original packet rate at a fixed bit rate. To maximize the original packet rate under this condition, the sum of fragment lengths needs to be minimized for each original packet.

In a compliant network, no IPv6 packet can be fragmented into less than two fragments, the first fragment never shorter than 1280 bytes. At the same time, the second fragment can't be shorter than the IPv6 header plus the Fragment Header plus one byte because a valid fragment can't be empty. Assuming the Fragment Header length of 8 bytes, the minimum sum of fragment lengths will be this: $1280 + 40 + 8 + 1 = 1329$ bytes.

We won't delve into the pathological behavior options available to a malicious or errant IPv6 source. To name just a few, the source can be creating undersized fragments or inserting a Fragmentation Header in packets without really fragmenting them. However, the reassembly algorithm should be able to tolerate such mild anomalies as long as they make sense and stay within the boundaries of the protocol. Thus tells Postel's Law.

Then it is easy to calculate, say, the wrap-around period of a 32-bit Identification at a traffic rate of 1 Tbit/sec:

$$\frac{8bit/byte \times 1329byte/packet \times 2^{32}packet}{10^{12}bit/sec} \approx 46sec. \tag{3.6}$$

As easy will be the calculation of the maximum safe rate of fragmented traffic given the recommended reassembly timeout value of 60 sec [82, Section 4.5]:

$$\frac{8bit/byte \times 1329byte/packet \times 2^{32}packet}{60sec} \approx \frac{46}{60}\times 10^{12}bit/sec \approx 7.6\times 10^{11}bit/sec. \tag{3.7}$$

What conclusion can be drawn from this regarding the 32-bit Identification? Well, a few more bits of headroom wouldn't have hurt, but even this ratio between the traffic rate and the wrap-around period[2] is satisfactory because it concerns only traffic from a single source to a single destination rather than backbone traffic between many hosts at once.

The *KAME* IPv6 stack cut corners a bit when it came to generating unique fragment Identification values. Instead of generating a separate sequence of them for each Source-Destination pair, a single system-wide sequence was used. Originally it was based on a simple counter and now *KAME*'s follow-on in *FreeBSD* has switched to a pseudo-random number generator with a large period. Of course, this simplification can't but reduce the wrap-around period somewhat.

Fragment Offset will be relative to the beginning of the Fragmentable Part rather than that of the original packet, as shown in Fig. 3.18.

A nice byproduct of this decision will be that no IPv6 Fragment Offset can point into the Unfragmentable Part and, in particular, into the main IPv6 header. As we remember, the IPv4 Fragment Offset was relative to the beginning of the packet and that opened the attack vector where subsequent fragments would overlay the most critical fields in the IPv4 header including Protocol, Source Address, and Destination Address. No such attack is possible in IPv6, but we are going to make IPv6 fragmentation even more secure by eventually outlawing overlapping fragments of any kind.

[2] Strictly speaking, constant is the *product* of the traffic rate and the wrap-around interval because they are *inversely* proportional to each other.

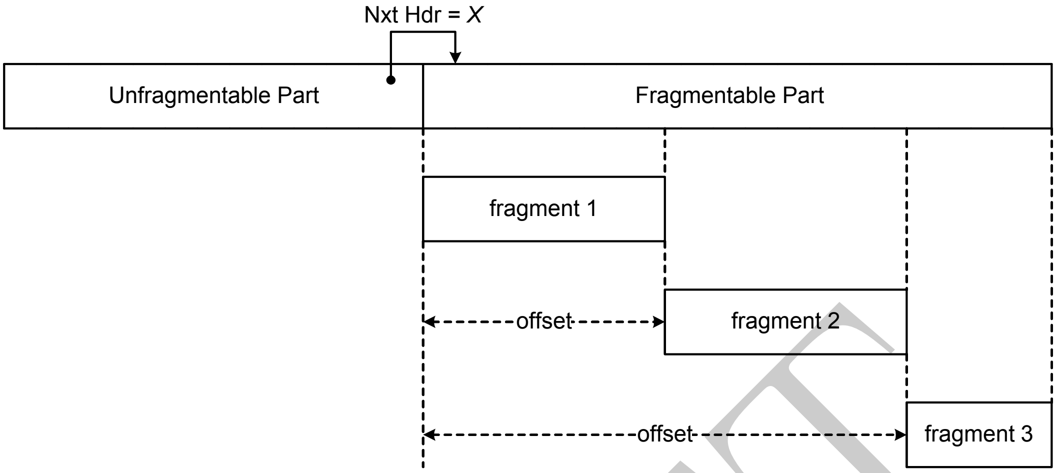


Figure 3.18: How IPv6 Fragment Offset is determined

The offset unit and the field length will stay the same as in IPv4: 8 bytes and 13 bit, respectively. Thus the actual fragment offset can be up to $(2^{13} - 1) \times 8 = 65528$ bytes inclusive. This range will be sufficient because no Fragmentable Part can be longer than the maximum IPv6 Payload, 65535 bytes, and the trailing 7 bytes will certainly fit in one fragment provided the Unfragmentable Part consists of just the main IPv6 header in order to maximize the Fragmentable Part.

> Keep in mind that IPv6 Payload Length includes the extension headers present in the packet.

The available fragment offset range turns out to be even larger than necessary: A sequence of fragments can be constructed to reassemble into a giant packet over 65575 bytes long. That only requires any fragment have the sum of its offset (in bytes) and raw length (less the fixed headers) greater than 65575 bytes minus the Unfragmentable Part length. (Consider Fig. 3.18 and Fig. 3.17 to see why it is so.) Should reassembly ever result in such a giant packet, it needs to be discarded and probably an error notification needs to be sent to its source.[3]

[3] Using the control message protocol from Section 4.3.

An exercise for the reader will be to work out the maximum achievable giant packet length assuming a PMTU of not less than 65575 bytes.

Solution guideline: Finish reading through this section. Then prove the following equation:

$$L = L_1 - F_1 - 8 + 8O_N + F_N,$$

where L is the IPv6 Payload Length of the original packet, L_1 is that of the first fragment, F_1 is the raw length of the first fragment in bytes, O_N is the offset of the last fragment in 8-byte words, and F_N is the raw length of the last fragment in bytes. Finally, maximize each term except F_1, which has to be minimized because of its negative sign: $\max L_1 = 65535$, $\min F_1 = 8$ (by the offset unit choice), $\max O_N = 8191$ (a 13-bit field), $\max F_N = 65535 - 8 = 65527$ (allowing for an 8-byte Fragment Header). The resulting giant payload length is quite impressive: 196574 bytes; and the complete packet is even longer than that by the 40 bytes of an IPv6 header! The sequence of fragments to reassemble into the giant-of-giants packet has to employ the trick with varying Unfragmentable Parts: The first fragment needs to have it inflated to the max, e.g., with *PadN* options, while the last fragment needs it minimized down to the bare IPv6 header. Needless to say, such a fragment series can be sent only by a curious researcher or malicious attacker because it places the destination host's robustness under a serious test.

The More Fragments, aka **M**, flag will have the same semantics as the IPv4 MF flag: It is set to 1 in all but the last fragment of a packet and to 0 in the last fragment.

The length of a raw fragment, prior to its encapsulation (Fig. 3.16), has to be a multiple of 8 bytes by the offset unit choice unless it is the last fragment. Otherwise the offset of the following fragment won't be a multiple of 8. Should the final destination receive a fragment with $M = 1$ whose raw length (Fig. 3.17) isn't a multiple of 8, this will need to be handled as a fault. Such a fragment will have to be dropped, probably with an error notice sent to its source.

Of course, when we say, "the first fragment", or, "the last fragment", we are referring to their position within the original packet, not their order of arrival at the final destination. The IP reassembly machine mustn't care about the order the fragments arrive in because the network may have reordered them in a completely random and unpredictable manner.

Now we can finally construct the Fragment Header as shown in Fig. 3.19 [82, Section 4.5]. The 8 bytes of its total length leave enough room for nice

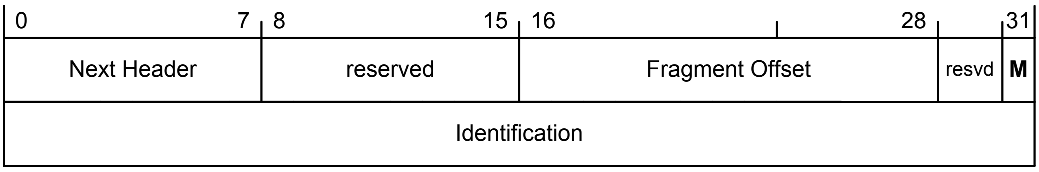


Figure 3.19: Fragment Header format

alignment of its fields. In particular, the placement of **Fragment Offset** is such that its value can be easily converted to byte units as follows: Read the whole aligned 16-bit word and then clear its three low bits by AND'ing with `0xFFF8`. This is equivalent to shifting the **Fragment Offset** value 3 bits left, that is, multiplying it by 8.

The IPv6 reassembly procedure is essentially the fragmentation procedure run back to front with safety checks added:

1. Having received a fragment, the reassembler extracts the {Source, Destination, Identification} tuple from it and uses the tuple as the key to search for an existing reassembly buffer in its memory.

    (a) If no existing buffer is found by the tuple, a new one needs to be allocated because it is apparently a new packet.

2. If **Fragment Offset** is zero:

    (a) Its Unfragmentable Part is saved to become that of the reassembled packet.

    > An IPv6 host implementation can choose to store the Unfragmentable Part in the reassembly buffer itself rather than in an external data structure associated with the buffer. However, in step 3 it will have to adjust the effective fragment offset into the buffer by the Unfragmentable Part length because the offset as indicated by **Fragment Offset** is relative to the end of the Unfragmentable Part. Furthermore, the reassembly buffer will have to have an extendable start boundary because the Unfragmentable Part length won't be known until the offset-zero fragment arrives and it may not be the first to come due to packet reordering by the network.

    (b) The **Next Header** value from the Fragment Header is saved as well to fix up the Unfragmentable Part.

The RFC [82, Section 4.5] tolerates the case where fragments of one original packet contain different extension headers and even **Next Header** values although such a discrepancy can be quite suspicious in practice. Which headers and field values will make it into the reassembled packet then? Precedence is the first fragment's here, i.e., the one's with an offset of zero, not to come first.

3. Decapsulate the fragment and put its raw data at the offset into the buffer as indicated by **Fragment Offset**.

4. If it is the last fragment ($M = 0$), the reassembly buffer length is "frozen" so that no fragments can extend beyond the last one.

5. If all blank space in the reassembly buffer has been filled in, the packet has been reassembled and now needs only a few final adjustments:

   (a) The Unfragmentable Part needs to be prepended.

   (b) **Payload Length** in the main IPv6 header needs to be set to that of the resulting packet.

   (c) **Next Header** in the last header of the Unfragmentable Part needs to be restored to the value saved from the Fragment Header in step 2b.

The reassembler, as it goes, needs to detect certain error conditions as we discussed above in this section:

- reassembly timeout so that incomplete packets don't linger forever;

- a fragment with $M = 1$ and a raw length that isn't multiple of 8;

- a reassembled giant packet whose total length is over 65575 bytes.

A natural reaction to such error conditions will be to drop the offending fragment or packet and, when safe, to send an error notification to its source.

We said, "when safe", because it won't be safe to return a notification for, e.g., a multicast packet or another notification. We will discuss this point in greater depth when it comes to ICMPv6 in Section 4.3.

Furthermore, there can be anomalies related to the fragment offset and status, namely:

- more than one fragment with $M = 0$;
- overlapping fragments.

To trigger these anomalies, they need to be different fragments, not mere duplicates of the same fragment brought about by the network.

At best, such anomalies can be caused by a collision between the key tuples of different fragmented packets, which is fraught with fragment misassociation and data corruption.

However, a worse, and likelier, cause will be a premeditated attempt to circumvent the network security policy. Here is a trivial example of this sort of attack. Suppose there is a web server protected with a basic packet filter only letting TCP packets to port 80 through. An attacker sends the first fragment including the TCP header as though its TCP Destination Port is 80, and so the packet is let through OK by the filter. The next fragment constructed by the attacker overlays the TCP header and changes the Destination Port to, say, 22 in the web server's reassembly buffer. The end result will be that the malicious packet is delivered to the socket on TCP port 22 not 80 notwithstanding the packet filter policy. Other transport-layer fields can be disguised likewise.

All in all fragment overlap can bring on a whole cluster of issues for no benefit, so the right choice will be to prohibit it altogether in IPv6 [107]:

- A compliant source shall never originate overlapping fragments.
- A final destination shall silently discard the whole packet being reassembled as soon as fragment overlap is detected in it. For complete protection, further fragments of this packet to arrive within the reassembly timeout window must also be silently dropped.

Consider other attack vectors leveraging IPv6 fragmentation. For example: An attacker sends two fragments back to front: the second fragment followed by the first one. The second fragment has an offset and length that don't qualify it as coming from a giant packet, and so the final destination allocates a reassembly buffer and waits for the first fragment. Then comes the first fragment, with its Unfragmentable Part pumped up to the max with dummy extension headers. These fragments when combined make up a giant packet that extends the reassembly buffer from its head due to a large Unfragmentable Part. This corner case can easily crash an unsophisticated implementation. Granted, this kind of attack will need a large PMTU.

3.3.5 IP Security Headers

The subject of IPv6 extensions headers provides us with a nice context to have a really quick look at **IPsec** (IP security). We need to because IPsec support is recommended in all IPv6 nodes [108, Section 11.1].

> IPsec support used to have mandatory status in the previous revision of IPv6 Node Requirements [109, Section 8.1]. Its demotion to recommended status in the current revision of the Node Requirements document [108, Section 11.1] doesn't mean though that IPsec fell out of favor. There were two quite practical reasons behind the relaxation of the IPsec support requirement [108, Section 11]: Firstly, for all its strengths, IPsec isn't a cure-all and it has no monopoly on security in TCP/IP. There are cases where security at a different layer will be preferable instead, such as TLS at transport layer or SSH at application layer. Secondly, the mandatory status of IPsec would be a serious obstacle in the way to IPv6 compliance for embedded implementations running on resource-constrained platforms such as coffeemakers or sensors. A complete IPsec module can be well beyond what their computing resources can handle, and without one their IPv6 stack would be rated non-compliant. This conflict has been resolved in the current IPv6 Node Requirements.

Of course, there is much more to IPsec than just its extension headers: IPsec is an architecture, that is, a complex system of models, mechanisms, and protocols aimed at end-to-end protection of IP packets. Its end-to-end quality means that only the source and the final destination are involved in it, e.g., as follows: the source signs the packet and the final destination verifies the signature; the source encrypts the packet and the final destination decrypts it. At the same time, intermediate nodes, whether routers or non-final destinations, handle protected packets just the same way as plain ones.

Historically, IPsec made its first appearance to protect specifically IPv6 and only later was it retrofitted to IPv4 as well, so its connection with IPv6 has remained quite strong. We won't digress now into the numerous details of IPsec merely because that would make for another book, and a rather big one. Instead, we are going to focus on the features of the IPsec headers that make them part of the IPv6 extension header family.

Among the IPsec protocols, directly interfacing with IPv6 are **AH** (Authenticated Header) [95] and **ESP** (Encapsulating Security Payload) [110]. The former offers only packet authentication and integrity protection while the latter also provides for encryption of packet payload and select header fields. By and large, ESP can do everything AH is capable of and even more than that, so there is little point in using them together. Let's see if we can work out the most essential requirements to their header formats from the first principles.

According to some sources, AH was created in addition to ESP only to overcome the US restrictions on export of cryptography [111]. The loophole was that pure authentication was exempt from the export restrictions although its mathematical basis wasn't too different from that of encryption.

What fields will be required in the AH header? First of all, by the very purpose of AH, there will need to be a sort of digital signature in it, referred to as **ICV** (Integrity Check Value) in the IPsec lingo. Both security algorithms and ways to attack them are in constant development, spurring each other on, so we should fix neither the **ICV** length nor its calculation algorithm. Consequently, the AH header will have to be of variable length, thus needing a **Length** field. Unlike the other IPv6 extension headers, the AH length value is in 32-bit units.

Similarly to the other IPv6 extension headers we have seen so far, the upper bound of the AH length range is extended by taking away a constant length from its value prior to storing it in the header; namely, two units are subtracted [95, Section 2.2]. Of course, it would seem more natural to take away the full length of the fixed part, which is, in fact, three units. On the other hand, taking away just two units ensures that the resulting AH length value will never be zero in a valid packet. There is certain prejudice against zero values in protocol headers based on the meaning of zero as "no data". Recall, e.g., the semantics of the **Checksum** field in UDP. Another reason for the particular choice of the value to be taken away is that the total length of an AH header needs to be a multiple of 64 bits in order to maintain the alignment of the subsequent data [95, Section 2.2].

The AH length field name is **Payload Length** [95, Section 2.2]—probably to confuse the Russians.

How are the security parameters to be conveyed, such as the algorithm chosen and its arguments, e.g., the secret key? It is assumed by the IPsec model that this information has already been made available to both the source and the destination in the form of a resident record known as a **Security Association** (**SA**) [112, Section 4]. If IPsec were a spy film, the codebook exchange would have been completed off-screen just because doing it this way is more secure. For our current purposes, it can be assumed that the SA was entered by hand in both hosts involved prior to sending any secure traffic.

> Of course, in reality SAs are seldom typed in by hand because it is far more convenient to have them managed by an auxiliary protocol such as IKE.

So, when it is time to actually send a packet through, the source can provide just a reference to the corresponding SA so that the destination can handle the security header properly. Spies would use the codebook page and line numbers for this purpose, and IPsec relies instead on a 32-bit **Security Parameter Index** (**SPI**) to identify the SA between the source and the destination. Only the SPI therefore needs to be stored in the eponymous field of the AH header to indicate which SA this packet is to be handled by [95, Section 2.4].

Next, to preclude a replay attack via sending an exact copy of the secure packet, the AH header will include a **Sequence Number** that increases by one for each packet sent through this SA [95, Section 2.5].

> At first glance, the basis of a replay attack is that some messages aren't idempotent. For example, by presenting two copies of the same 1-dollar check we can get 2 dollars in cash; that's why checks usually carry a serial number on them. On the other hand, just repeating the signal to raise a railway semaphore arm to its "clear" aspect won't have any effect because the arm is already in its upper position after the initial signal. Nevertheless, the attacker can save an image of the signal and replay it later on, when the semaphore is in its "stop" aspect, thus opening the semaphore and bringing about a train crash. So it is unsurprising that a secure data exchange system must be able to withstand attacks of this class regardless of whether it involves non-idempotent signals or not.

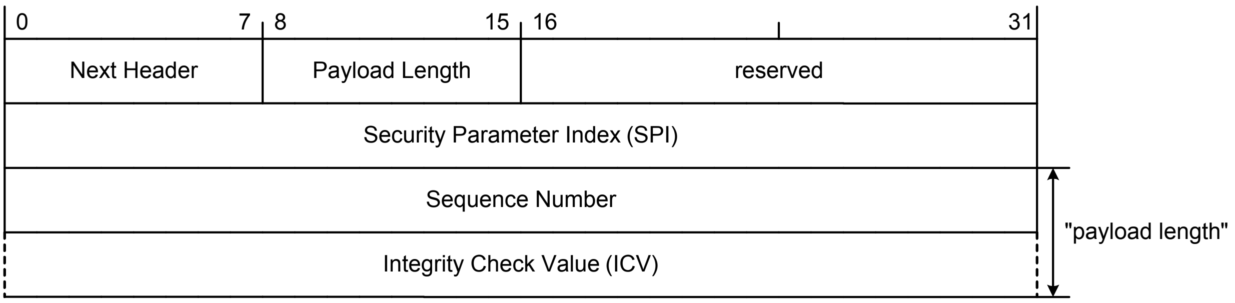


Figure 3.20: AH format

> Packets originally sent in order can be randomly reordered by the network, and so the destination can't expect **Sequence Number** to be increasing monotonically in the packets received. For this reason, a sliding window has to be employed to verify **Sequence Number** values [95, Section 3.4.3]: If the window is thought of as sliding to the right, packets out and on the left of the window are just discarded; packets falling within the window are checked against a list of packets already seen within the window; and the reception of a packet out and on the right of the window advances the window along the Sequence Number axis.

Lastly, the AH is nothing else but yet another extension header, to be followed by more extension headers and/or upper-layer protocol payload. Consequently, there will need to be a **Next Header** field in the AH format for the destination to be able to interpret the subsequent data. As confirmed by [95, Section 2], we have just managed to reconstruct the complete AH format from the first principles. The end result is shown in Fig. 3.20.

For the sake of completeness, it needs to be pointed out that protected by ICV will not only be the packet's tail, starting from the AH, but also as much of the preceding headers as possible. The problem is, some fields in those headers can change in transit, conflicting with the end-to-end security requirement that protected data be immutable. If we can identify all of such fields, the ICV computation will be able to skip them or, better, substitute zeros byte-for-byte. Zero substitution is preferable here because it will provide at least for field length protection as long as the ICV algorithm is sensitive to the *number* of zero bytes appearing in succession. Now back to the identification of mutable fields. For IPv6 options, this problem was already solved in Section 3.3.2 by having a dedicated bit in the option type provide the information as to the option mutability. IPv6 extension headers, unlike options, are supposed to be completely standardized and so they can be categorized by their mutability once and for all in the correspond-

Table 3.2: Mutability of IPv6 header fields

Immutable	Mutable but predictable	Mutable and unpredictable
• **Version** • **Payload Length** • **Next Header** • **Source Address** • **Destination Address** if no Routing Header (Type 0) present	• **Destination Address** if a Routing Header (Type 0) present	• **Traffic Class** • **Flow Label** • **Hop Limit**

ing RFC document [95, Appendix A2]. Here we may want to use a ternary classification rather than a binary one as follows:

- immutable;
- mutable but predictable;
- mutable and unpredictable.

The new category, “mutable but predictable”, is for headers and fields whose ultimate value to be seen by the final destination can be reliably predicted by the source. In reality, the only components classified so are RH0 and the IPv6 **Destination Address** when RH0 is present.

Now let’s complete the taxonomy by categorizing the main IPv6 header fields as shown in Table 3.2.

As for the IPv6 extension headers, there is effectively nothing left to be done about them as far as their mutability goes: For the Option Headers, the problem has been solved on per-option basis; the predictably mutable RH0 has been deprecated; and the Fragment Header can never be even considered for protection because IPv6 fragmentation is to act on a complete packet already protected with AH and reassembly occurs before AH is verified, so IPsec can never encounter a Fragment Header [95, Appendix A2] unless something went wrong in the stack.

> A fragment packet can enter a secure tunnel, but then it will be subject to tunnel encapsulation and the resulting tunnel packet to be passed on to IPsec won't be a fragment. This is as applicable to IPsec tunnel mode, which only amalgamates the conceptual stages of tunneling and protection into one handy service.

Now on to ESP. A new task for us will be to conceal the packet contents using encryption. In order to hide as much information as possible, ESP needs to encrypt not only the upper-layer protocol data following the ESP header but also the clue about its meaning, that is, the **Next Header** value. Then in theory an eavesdropper will have to make a blind guess on what kind of protocol data is encapsulated in the secure packet. But in practice the Next Header field hardly enjoys a variety of values as most application packets now carry plain TCP (6) or, considerably less often, UDP (17) while tunnel packets encapsulate IPv4 (4) or IPv6 (41). Should the encryption start with Next Header, a predictable beginning of the cleartext will open a vector for a sophisticated attack on the ciphertext because common encryption algorithms work sequentially through the input data, byte by byte or block by block.

> Even CBC mode can be made vulnerable by a predictable cleartext beginning because the first block is chained with none.

To block this attack vector, the Next Header value will be placed *after* the upper-layer protocol data so that it gets encrypted last and its ciphertext has a chance to vary depending on the preceding data [110, Section 2]. Next to it will be the **Padding** and **Pad Length** fields required for block cipher support. So the ciphertext will be preceded only by the **SPI** and the **Sequence Number**, both having to come in cleartext by their role.

If the Security Association requires that packets be signed and authenticated in addition to encrypted, **ICV** can be appended after the ciphertext so that an eavesdropper having no access to the SA can't even know if the packet is signed.

ESP owes its name to this secure packet structure, shown in Fig. 3.21: Unlike AH, it doesn't just insert itself into the extension header chain but instead encapsulates the data to be protected, thus making it opaque to an eavesdropper.

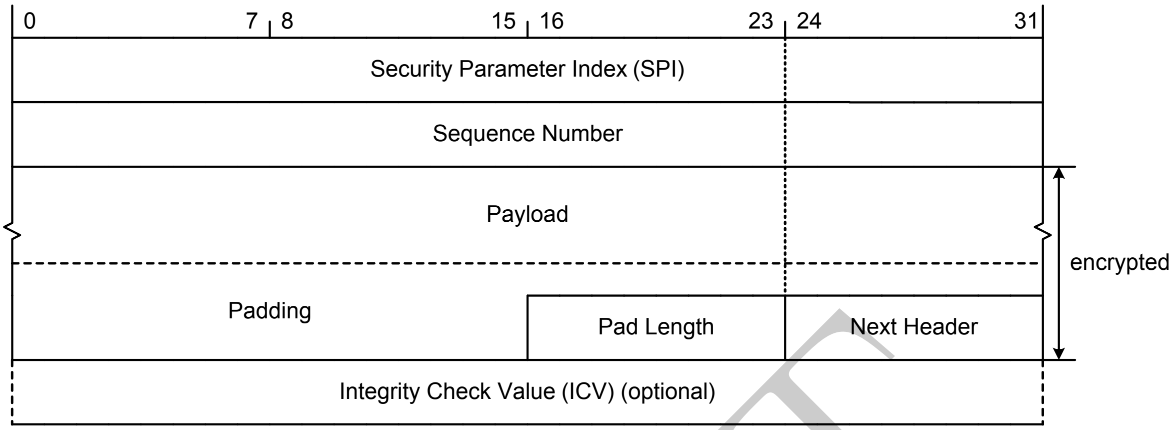


Figure 3.21: ESP encapsulation format

Although denied direct access to the packet data, a sophisticated attacker can still be conducting reconnaissance by the study of ESP packet size and timing between the target nodes. Fortunately, complete payload encryption by ESP provides for blocking this side channel as well. Any variation in application packet size can be masked with appropriate padding [110, Section 2.7]. As for packet timing, it can be scrambled by inserting dummy ESP packets at random intervals. "No Next Header" from Section 3.3.1 will come handy in constructing such dummy packets. Together these simple yet effective tricks provide for **traffic flow confidentiality** (**TFC**).

ESP being well-isolated and self-sufficient has a downside: Unlike in AH, ICV no longer protects the preceding IP header [110, Section 2, Section 3.3.2]. For this reason, the main IP header can't be blindly trusted in an ESP packet. On the other hand, there aren't too many ways to tamper with an ESP packet while still keeping it valid. Provided the SPI selected a valid SA leading to successful authentication and decryption of the packet payload, the destination can conclude the following about the packet:

- The payload is authentic.
- The packet came from a trusted source.
- The SA *may* identify the real source of the packet.

The "may" is due to the IPsec architecture permitting but not requiring an SA to be bound to specific source and destination addresses [112, Section 4.4.1].

Nevertheless, limited attacks can still be mounted on protocols sensitive to exact source and destination address values. For instance, spoofing those addresses can allow to insert segments from one TCP session into another provided the TCP sessions have different addresses but equal ports.

> As we know, a TCP session is identified by the {*address1*, *port1*, *address2*, *port2*} tuple. Therefore there can be two concurrent and distinct TCP sessions differing in, say, *address1* only.

This attack vector can be blocked by turning the IP header into mere payload before ESP gets to the packet. This sort of transmutation is easily done by means of tunnel encapsulation. If we limit our tunnel protocol choice to IP-IP only and then let IPsec onto the tunnel packet, we will have what is known as **tunnel mode** in the IPsec lingo. It is as applicable to AH, although its importance to ESP is greater due to no outer IP header protection provided by the latter. The other IPsec mode, which boils down to direct protection of application packets, is known as **transport mode**, probably because Layer 4 protocol datagrams will be the majority among the packets protected if one opts for this mode.

> Of course, other tunnel protocols such as GRE or L2TP can be protected by IPsec equally well, for IPsec can protect anything IP.

> The reader will sketch the structures of AH and ESP packets in tunnel mode. What Next Header values will be encountered in them? Consider IPv4 and IPv6.

Now that address spoofing was fended off, we can take time to ponder a legitimate case where essentially the same issue can hit. Who, or what, else can alter an IP header apart from attackers? Of course, NAT boxes can—and will. There are scenarios where NAT can be a useful tool, but its problem is that it can't go without changing IP header fields. While ESP can tolerate address translation in the outer IP header, AH will consider the translated packet fake and drop it.

The reader may point out to NAT-T as a means to make IPsec and NAT get along. However, in reality NAT-T can solve only a rather specific problem of getting IPsec through a "masquerade" NAT box mapping many internal IP source addresses to a few external ones. This kind of NAT results in loss of addressing information at IP level and the NAT box tries to make up for it at transport level by effectively stashing the info bits otherwise lost, in Claude Shannon's sense, in the TCP and UDP port numbers or other suitable fields such as ICMP Echo ID. To make this scheme applicable to IPsec, the latter has to be encapsulated in UDP. It isn't too hard to see that this trick has no chance to solve the fundamental conflict between AH and NAT, so it is no surprise that NAT-T is defined only for ESP [113].

3.3.6 Extension Header Ordering

As we have worked out in this chapter, an IPv6 packet can contain multiple extension headers, with different header types playing different roles. So it won't be unreasonable to suppose that the relative header order may be significant. At the same time, the header chain format we saw in Fig. 3.2 (page 94) places little restriction on the number or order of headers. Indeed, each IPv6 extension header has a **Next Header** field in it, so the packet source is still free to put the headers chosen for the packet in any order and even to insert multiple headers of the same type. Our concern is that at least some permutations of headers may not make sense. Therefore additional rules need to be worked out now regarding the order of headers in the IPv6 packet. Fortunately, all the legwork has already been done, so all we have to do now is draw a few conclusions from what was already said.

We will start by pointing out that there is just one extension header type of interest to all nodes along the IPv6 packet's path: Hop-by-Hop Options Header. One such header can carry multiple options in it, so there is little reason to include more than one instance of this header in the packet. So we can reaffirm that the Hop-by-Hop Options Header can only be second to the main IPv6 header. This rule will simplify an IPv6 router's job as the router will never have to look beyond the first extension header.

Another two header types will be of interest to intermediate destinations: Destination Options Header and Routing Header. A Routing Header having a non-zero **Segments Left** value signals to the current destination that the packet is still to be forwarded on. Therefore the current destination doesn't need to look beyond the Routing Header. Consequently, the Destination Options intended for the eyes of all destinations need to precede the Routing

Header.

The rest of the extension headers and the payload are meaningful to the final destination only. If the packet needs to be fragmented, they will form its Fragmentable Part, get split across fragments, and so become generally inaccessible to intermediate nodes. Among those extension headers, the IPsec headers stand out as capable of protecting the remaining data. In particular, encryption by ESP can hide the data from the prying eyes without interfering with the IPv6 protocol. So the most natural position for the IPsec headers will be in front of the Fragmentable Part.

> This placement of the IPsec headers will also guarantee the immutability of the subsequent data en route, which is prerequisite to the applicability of IPsec.

Lastly, we assumed in Section 3.3.4 there would be options important only to the final destination. Such options can go into a separate Destination Options Header placed after the IPsec header and protected by it, e.g., encrypted.

The Fragment Header is the last one to pin down, but this has effectively been done: It will appear just after the Unfragmentable Part, i.e., following the Routing Header.

No extension header is mandatory in an IPv6 packet, so the header order we have just worked out is relative rather than absolute. Let's sum it up [82, Section 4.1]:

- Main IPv6 header (mandatory);
- Hop-by-Hop Options Header (optional);
- Destination Options Header for all destinations (optional);
- Routing Header (optional);
- Fragment Header (optional);
- IPsec header (optional);
- Destination Options Header for the final destination (optional);
- upper-layer protocol data (optional).

Thus each extension header should occur at most once in an IPv6 packet, except for the Destination Options Header which should occur at most twice [82, Section 4.1].

We will conclude this section by pointing out that this header order provides for sequential processing of headers, the current node never having to look ahead or behind.

> Deviation from the specified header order can have unwanted side effects. For example, should the Routing Header be found after the Fragment Header, the current destination will forward the packet by its Routing Header only after having reassembled it. This will not only violate the basic rules of IPv6 fragmentation and reassembly but also disrupt the operation of PMTUD because the source fragmented the packet so that it could get all the way through the path and the current destination has effectively undone its work. Of course, in this scenario it is the source's fault in the first place that it put the headers the wrong way. But there can be cases where tricks on the header order are played for malicious purposes, so the packet destination needs to take care when working through the header chain. Unfortunately, the reference *KAME* stack doesn't verify the header order in incoming packets, blindly processing any headers present one by one. The standard, in its turn, only recommends the header order and doesn't require it be enforced. [82, Section 4.1] even goes as far as requiring that the node accept and try processing extension headers in any order and occurring any number of times in the same packet. We can't but suggest this aspect of IPv6 may need a review.

3.3.7 Unsupported Header Handling

The modular structure of the IPv6 packet suggests that new extension header types can be introduced as IPv6 evolves. But as soon as a new header type appears, the older nodes will be encountering a Next Header value they know nothing about. What will the safest way to handle such an exception be? Will the current node be able to signal the problem back to the source? Or will it manage instead to skip over the unsupported header? Let's take time to consider these issues before we can call our work on the IPv6 packet done.

First of all, we need to see which node types can encounter an unsupported extension header. It can hardly be the source as the packet is its own creation and so it is supposed to understand why each header there. Regular routers are not to look beyond the Hop-by-Hop Options Header, which they must support, so they will never delve into the packet far enough to encounter an unsupported header. As for the current destination, it will need to scan through the IPv6 header chain until it reaches a Routing Header or payload and in the process it will indeed have a chance to run into an unsupported header.

Aside from the conventional IP node types, there can be various "middleboxes" in the network, such as firewalls and intrusion prevention systems. A middlebox is usually disguised as a router or even a bridge but, unlike the latter two, it does deep stateful analysis of the transit traffic, keeping track of application sessions rather than handling transit packets in a forward-and-forget manner. This complexity enables middleboxes to enforce security, detect protocol anomalies, and even fiddle with protocol data units, NAT being one case where the latter is required. To be rated as IPv6-ready, a middlebox will have to be able to work all the way through the header chain so as to check its compliance with the configured policy as well as to get to the payload. No doubt, an older middlebox will have trouble understanding a newly introduced extension header.

Next, we are going to verify whether skipping over an unsupported header is possible. Consider how an IPv6 packet is parsed. The high-level algorithm, as long as its gory details can be omitted, is rather simple:

1. Start from the main IPv6 header, known to be at offset zero.
2. For the current header:
    (a) Locate the **Next Header** field and fetch its value.
    (b) Work out the length of the current header in order to locate its end.
    (c) Move on to the next header, which is known to be right after the current one.
3. Repeat item 2 until the next header is an upper-layer protocol header.

Knowledge of the current header format is prerequisite to items 2a and 2b. Indeed, different extension header types encode their length differently and place the **Next Header** field at different offsets. This is an obvious and serious but not the most fundamental obstacle to skipping over an unsupported header.

The most basic obstacle here lies in item 3. Suppose the current header itself is supported but the **Next Header** value found in it makes no sense to the node. Since the values for IPv6 headers and upper-layer protocols come from the same IP protocol registry (see Section 3.1), the node can't even know whether it is an extension header or a payload that follows the current header. The problem is due to the IP protocol registry having no structure, so a new value from it, say, 254, can indicate an experimental transport protocol as well as a new extension header.

The inability to skip over an unsupported header isn't really a problem because it is consistent with extension headers' role of carrying mandatory instructions to nodes. This is what sets extension headers apart from IPv6 options.

Therefore a destination that encountered an unsupported header has no choice but to discard the packet and perhaps to send an Unrecognized Next Header control message back to the source.

Our customary "perhaps" was due to there being cases where sending back a control message could be unsafe. E.g., it would generally be a poor idea to respond this way to another control message or to a multicast packet. We will discuss this matter in detail when speaking about the control message protocol for IPv6 in Section 4.3.

And what should a middlebox do when in the same situation? It won't be able to perform deep analysis of the packet, so it is up to its current policy to tell the box what to do. If the policy is liberal, the packet can be let through hoping for the better; but if the policy is restrictive, the packet will have to be dropped.

Note how the behavior of the destination was defined by the protocol while that of the middlebox was dictated by a policy.

Let's recap the difference between a protocol and a policy. A protocol is what enables different parties to communicate with, and unambiguously understand, each other while a policy governs the manner their conversation is to be conducted in. For example, English is a protocol but the way to address a member of the British royal house is a policy. Apart from that, a policy can tell how adjustable protocol parameters are to be set. E.g., it can require TCP MSS be reduced to a certain value—or prohibit speaking loudly after lights-out.

To provide for a more flexible handling of unrecognized headers by middleboxes, it was proposed to encapsulate newly introduced headers into a TLV-based container header defined in advance so that everyone supported it [114]. It isn't quite clear to us though why this mechanism may be needed when there are IPv6 options based on exactly the same idea.

Chapter 4

IPv6 in the Protocol Stack

4.1 Link-Level Encapsulation

Nominally, IPv6 is yet another version of IP that can freely coexist with IPv4 thanks to the **Version** field in its packet header. Therefore IPv6 per se requires no changes in link-level encapsulation procedures unless those have ties to IPv4.

Take for example the SLIP protocol [115]: It has been IPv6-ready since well before IPv6 was born. Furthermore, SLIP can alternate packets of any IP versions in the same link because the receiver can always detect the version of the incoming IP packet by the first nibble in its header.

As we remember, the inner protocol type is conveyed implicitly in SLIP, by programming or configuring it in both nodes before the link can operate. This is why a SLIP link can't carry more than one Layer 3 protocol at the same time. Very often the protocol type is just hardcoded to IP in the SLIP driver and can't be changed at run time, but that is none of the SLIP protocol's inherent limitations: A SLIP line is perfectly able to carry any single Layer 3 protocol as long as its ends agree on what it is. For example, almost two decades back the future author of this book wrote a proof-of-concept *MS DOS* driver to encapsulate IPX into SLIP and had barrels of fun playing *Doom* duels with his pals over the quite OSI-compliant link.

By the way, SLIP makes for a nice case in the holy war over whether IPv4 and IPv6 are just different versions of the same protocol or distinct protocols: If they were different protocols, they would be unable to operate simultaneously through the same SLIP line. The opposite being the case, they have to be regarded as different versions of the same protocol.

4.1.1 Ethernet

We might just as well have said that the well-known procedure for transmission of IP over Ethernet [116] would be applicable to any IP version. However, the case of *Ethernet* proves not as trivial as that of SLIP.

To start with, the more ubiquitous *Ethernet* enjoys a greater number of implementations out there that are "slightly broken". In particular, it was discovered that some "Layer 4 switches" boasting wire-speed support for IP, TCP, and UDP[1] in addition to *Ethernet* would completely overlook the tiny IP **Version**, assuming that an **Ether Type** of `0x800` always meant IPv4 [117][118, Section 9.8.1]! However ugly, this quirk justifies a separate **Ether Type** value for IPv6: `0x86DD` [121, Section 3].

In her recurring lecture [119, 120], Radia Perlman complains that the RFC text on IPv4 [85] and IPv6 [82] fails to require the IP Version be checked *on input*. We tend to agree with Ms. Perlman that it would, by the modern standards, be a serious flaw on the protocol developers' part. To be completely fair, however, we need to put this issue in its historical context. The fact is, for a long time the Internet standards were written for, and by, top experts of the industry, who had no problem translating the wire data format to a robust and secure receiver algorithm. So it is no surprise that the terms "protocol" and "packet format" remained interchangeable through a whole period of TCP/IP history. Alas, today the average RFC reader's level is nowhere as high as it used to be in the 1970s, which has to be made up for by more rigorous and comprehensive standards documents leaving no room for ambiguity.

Granted, a protocol can be so complex that providing only its wire data format will no longer be sufficient; but the failure of some implementations to check Version in incoming IP packets speaks of certain ills in the prevalent technical culture rather than of this protocol's complexity.

Discuss what the best *input* procedures would be to handle an IPv6 packet received with an Ether Type of `0x800` and an IPv4 packet that came with an Ether Type of `0x86DD`. (Hint: *Apply Postel's Law to find an optimal answer.*)

Next, there is more to *Ethernet* encapsulation of IP than just an Ether Type choice. It also concerns the way IP multicast addresses are mapped to their MAC-48 counterparts: multicast address resolution. A really naive solution would be to use ARP or a similar mechanism for the purpose, with the

[1] Whatever the meaning of such claims by the device vendor be.

multicast source broadcasting a request first, listening to the replies, and thus learning the list of group members' MAC addresses. However, this solution isn't really going to cut it because IP multicast is instrumental in network resource discovery. Having it rely on ARP or another network-based address resolution mechanism would mean doing discovery in order to do discovery—a sure way to a chicken-and-egg situation where a packet needs to be sent only to request information required to send that packet. For this reason, IP multicast has to stick to the following principle: *IP multicast address resolution mechanisms must be local.* In other words, a multicast source needs a means to map an IP multicast address to the corresponding MAC address without ever having to ask other nodes. As long as this principle is met, an outgoing IP multicast packet can be passed down to the link layer immediately, based solely on the information already contained in it.

As we may remember, IPv4 easily met the above principle by employing a trivial bitwise operation on the multicast destination address: 23 least significant bits of the address were OR'ed into the corresponding low-order bits of the well-known MAC multicast address *01-00-5E-00-00-00* [64, Section 6.4]. It wasn't a one-to-one mapping because there were as many as 28 significant bits in an IPv4 multicast address, the high-order 4 always occupied by the well-known "Class D" prefix of `1110`. Because of that, an IPv4 multicast listener could receive other groups' traffic at link level if different IPv4 multicast addresses present on the LAN mapped to the same MAC address. To work around an insufficient selectivity of the mapping, the IPv4 multicast listener had to filter incoming packets by their IPv4 destination addresses, accepting a multicast packet only if the destination group was actually listened to on this ingress interface.

There is no reason to invent an entirely new mechanism of multicast address resolution for IPv6, but we will need to adjust a few parameters in the existing one to optimize it for the new environment. There are incomparably more IPv6 multicast addresses than their IPv4 counterparts, so it will be a good idea to extend the bitwise mapping to keep conflicts at bay. As many as 32 low-order bits will be transferred from an IPv6 multicast address to the corresponding MAC address. Consequently, a different, 16-bit, MAC-48 prefix will be required. For this role *33-33* has been chosen, resulting in the MAC-48 address range of *33-33-00-00-00-00* through *33-33-FF-FF-FF-FF* used by IPv6 multicast.

Verify that those MAC-48 addresses are multicast.

Notice that those are locally administered addresses. Unlike *01-00-5E*, the new OUI range *33-33-XX* didn't need to be purchased from *IEEE*.

Why `33-33`? 3333 Coyote Hill Road, Palo Alto, California was the address of Xerox PARC, where *Ethernet* was born [122, Section 2.3.1].

Discuss disadvantages of using locally administered MAC addresses in a well-known protocol. (Hint: *There can be address conflicts with in-house protocols, but Layer 3 address filtering should minimize their impact.*)

Here is an example of IPv6 multicast address resolution: The IPv6 addresses *FF02::C001:DEAD:BEEF* and *FF0E::BA1D:DEAD:BEEF* both map to the same MAC-48 address *33-33-DE-AD-BE-EF*.

Although such conflicts in IPv6 multicast address resolution are nearly harmless, only slightly affecting the overall network performance, their rate can be further reduced if non-overlapping group ID ranges are allocated to different classes of IPv6 multicast addresses by a policy decision, e.g., as follows [73]:

- permanent IPv6 multicast addresses: `0x0000 0000`–`0x3FFF FFFF`;
- permanent IPv6 multicast group identifiers: `0x4000 0000`–`0x7FFF FFFF`;
- dynamic IPv6 multicast addresses: `0x8000 0000`–`0xFFFF FFFF`.

Permanent IPv6 multicast addresses and permanent group identifiers are different in that the former are used in core IPv6 protocols while the latter are assigned to Internet-wide services. For example, `0x0000 0001` (All Nodes) qualifies as a permanent IPv6 multicast address whereas `0x4040 4040` (Network Time Protocol) is just a permanent group identifier.

This group ID allocation policy is also endorsed by the *IANA* registry of IPv6 multicast addresses [57].

As it is easy to see, none of the recommended group ID values exceeds 32 bits, precluding address resolution conflicts as long as this policy is adhered to. Nevertheless, an IPv6 multicast listener still must filter incoming packets, accepting only those to its groups joined on this interface.

4.1.2 PPP

An example of a well-known link-layer protocol that needs to be extended before it can support IPv6 is PPP. In order to provide its consumers with the convenient service of upper-layer protocol parameter negotiation, a PPP link is specifically prepared at establishment time to the operation of each Layer 3 protocol enabled on it. This can include negotiating Layer 3 addresses, netmasks, etc. The task is delegated by PPP to an auxiliary protocol of the NCP family, which naturally has to match the Layer 3 protocol and accommodate its features. In particular, the existing NCP for IPv4, IPCP [123], can only support 32-bit addresses, making it unsuitable for IPv6. Therefore IPv6 needs an NCP of its own: **IPV6CP** [124].

First of all, we need to assign well-known protocol numbers to IPv6 and IPV6CP from the PPP parameter registry [125] so that PPP frames with the new PDU types in them can be demultiplexed unambiguously. IPv6 will have an ID of `0x0057` and IPV6CP, by convention, will get the same value plus `0x8000`, i.e., `0x8057`. Now that both protocols can be encapsulated in PPP frames, we can move on to working out specific tasks for IPV6CP to fulfill.

Among all NCP tasks we can think of, which one will be the most specific to IPv6? By our plan laid out in Section 2.7, an IPv6-enabled network interface can benefit from an EUI64-like ID for its quasi-unique interface ID to be based on. The easiest case is when the interface has a genuine, globally unique EUI-64 on it. A MAC-48 address, if available, is no worse because it can be transformed into an EUI-64 using the well-known recipe from Section 2.7. But PPP itself employs no link-layer addresses at all because implicit addressing is quite sufficient for a point-to-point link: Any frame transmitted over it by the local node is received exclusively by the remote node and vice versa. Here we have a job for IPV6CP: to negotiate a pair of unique IDs for the PPP link endpoints. IPv6 will ultimately need a Modified EUI-64 on each end, so it makes sense to design the negotiation procedure so that it immediately produces IDs in this format, with no intermediate EUI-64 or MAC-48 involved.

Let's define the requirements to an ID pair negotiated so [124, Section 4.1]:

- Quite obviously, the two IDs must not be the same. They don't need to be globally unique though because they are supposed to be combined with a unique IPv6 subnet prefix assigned to this link.

> Of course, the resulting IPv6 addresses will have the same scope as the subnet prefix. E.g., a global prefix will make for globally unique addresses while the link-local prefix will provide for addresses unique only in the scope of this PPP link.

- The IDs should be reasonably stable, not changing across node reboots. Each node can use whatever sources of uniqueness are available to it when generating a tentative ID. A global EUI-64 or MAC-48 from another interface, if available, can make for the best choice. Otherwise hardware serial numbers can be used. As a last resort, a random ID can be tolerated.
- The Modified EUI-64 format must be adhered to. In particular, the **U/L** bit can be set to 1 only if based on a global EUI-64 or MAC-48 owned by this node—the standard transformation procedures from EUI-64 and MAC-48 to Modified EUI-64 were already defined in Section 2.7.

As for the interface ID negotiation procedure, it is nothing short of a nice and clear example of how PPP options work. The only message types required will be these: *Conf-Req*, *Conf-Rej*, *Conf-Nak*, and *Conf-Ack*—see Table 4.1. The corresponding IPV6CP option code will be 1 [124, Section 4].

> As we may remember, the negotiation of a PPP option goes in each of two possible directions independently of the other. For this reason, the IPV6CP peers conduct as many as two apparently independent conversations. One of them will yield an ID for node A to use and the other, for node B.

> The IPV6CP negotiation procedure allows for a scenario where node B initially advertises a zero tentative ID and node A picks not only an ID for itself but also one for node B. In this case, the **U/L** bit needs to be cleared in the ID offered to node B even if it was based on a global EUI-64 because the EUI-64 is still owned by node A not B. The **U/L** bit can be set to 1 only provided that node A somehow had a spare EUI-64 and now is leasing it to node B for its exclusive use until the end of the PPP session [124, Section 4.1].

To prevent the interface ID negotiation from entering an infinite loop, there will be a few extra rules for an IPV6CP party to follow:

Table 4.1: IPV6CP interface ID negotiation messages

Message	Meaning	Valid responses
The request		
Conf-Req	The message source wants to assign ID such-and-such (enclosed therewith) *to itself* (not to the other node!) Until acknowledged by the far end node, it is only a **tentative interface identifier**. By sending a tentative ID of all zeros, a node can indicate that it was unable to pick a reasonable ID, and request the other node suggest one for it.	*Conf-Rej*, *Conf-Nak*, *Conf-Ack*
Response options		
Conf-Rej	The message source doesn't support this option at all. The other node must send another *Conf-Req* without this option or terminate the session.	*Conf-Req*
Conf-Nak	The message source supports this option, but its tentative value as proposed by the other node was unacceptable: The tentative ID was zero or the same as this node was going to use for itself. This node therefore offers a new tentative ID *for the other node to use* (enclosed therewith). The other node needs to consider the ID proposed so and send a new *Conf-Req*.	*Conf-Req*
Conf-Ack	The message source indicates that it was happy with the other node's choice of interface ID because it was neither zero nor the same as the source's own tentative ID. The negotiation is complete.	None

- It shall never repeat itself, that is, send the same tentative ID value more than once in a *Conf-Req* or *Conf-Nak*.

> Nevertheless, it is OK for the ID value sent in a *Conf-Nak* to appear in the *Conf-Req* received in response. The non-repetition rule won't be violated in this case because the ID value will come back from the other party whereas the rule concerns the behavior of one party only, traditionally assumed by protocol documentation to be the local party.

- It shall not be picky about what its peer sent, accepting the first suitable ID it sees; namely:
  - If the tentative ID from the incoming *Conf-Req* isn't the same as the local one, it must be acknowledged with a *Conf-Ack*. Thus the local node approves what the remote node chose for itself.
  - If the suggested ID value from the incoming *Conf-Nak* isn't the same as the one sent by the local node in its last *Conf-Nak*, the local node must accept it as its new tentative ID and send a *Conf-Req* with it. Thus the local node readily accepts the first suitable suggestion from the remote node.

> Note how two seemingly independent IPV6CP conversations, one led by node A and the other by node B, prove linked with each other behind the scenes: For either node to respond to an incoming *Conf-Req*, it needs to consider its own tentative ID, to be advertised in an outgoing *Conf-Req*. Also, a compliant response to an incoming *Conf-Nak* is possible only provided the suggested ID value from the most recent outgoing *Conf-Nak* was kept in memory.

The interface ID negotiation process is drawn by these rules to one of two most probable paths:

- either each node immediately picks a unique ID for itself and the other node *Conf-Ack*'s it,
- or there is an initial conflict of IDs resolved when the nodes suggest different IDs to each other.

Of course, there can be more than one conflict in a row, but the probability of that is negligible unless we are dealing with broken implementations. In particular, a permanent conflict is possible if both peers pick the same tentative ID value and then use the same deterministic algorithm to derive new values from it, but we won't focus on this "pathological" case. Almost every modern computer system out there has a suitable source of uniqueness or randomness, and in the worst-case scenario it can solicit help from its peer by sending a zero ID value in its first *Conf-Req*.

> Should both peers send a zero ID to each other in their respective *Conf-Req* messages, the negotiation must be terminated with a *Conf-Rej* for robustness's sake [124, Section 4.1].

There can be a more pronounced impact on the negotiation process flow from how concurrent the two opposite conversations are. To get a better grip on the IPV6CP machinery, let's consider a zero-order process and a first-order process (i.e., an immediate acknowledgement and a conflict followed by accepting the peer's suggestion) in two extreme cases where the opposite conversations are either perfectly synchronous or fully consecutive.

Should both peers manage to generate unique values for their respective interface IDs from the outset, their IPV6CP conversations won't need to lock into each other and so the degree of their concurrency will make little difference: The end result will be just the same, each peer acknowledging the interface ID the other peer chose for itself. In this case, the consecutive scenario diagram in Fig. 4.2 can be obtained from the concurrent scenario diagram in Fig. 4.1 merely by shifting one of the conversations down the time axis.

By contrast, if the peers have accidentally chosen the same tentative interface ID, the negotiation outcome will depend on how the two opposite conversations are aligned on the time axis. When they proceed in parallel, each peer will go with the other peer's suggestion and make it its own interface ID as shown in Fig. 4.3. Alternatively, if those conversations are conducted in a consecutive manner, whichever peer starts its conversation first will have to accept the other peer's suggestion while the other peer will retain freedom to use its original tentative interface ID value as illustrated in Fig. 4.4.

> The reader will draw a similar pair of diagrams for the case where one peer solicits assistance in choosing an interface ID by advertising a tentative ID of all zeros in its first *Conf-Req*.

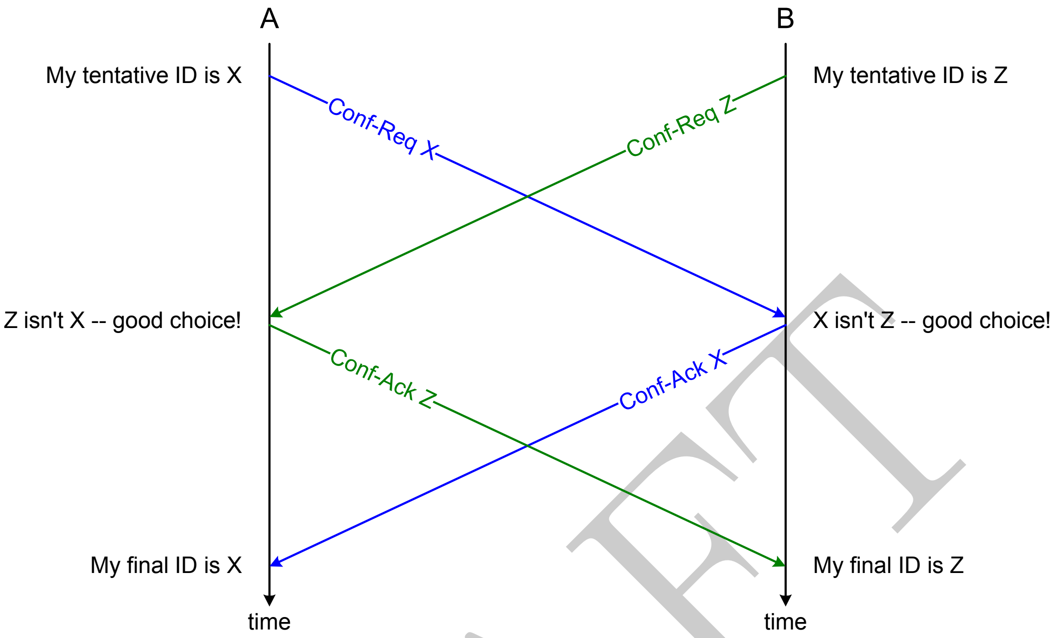


Figure 4.1: IPV6CP: concurrent negotiation without conflict

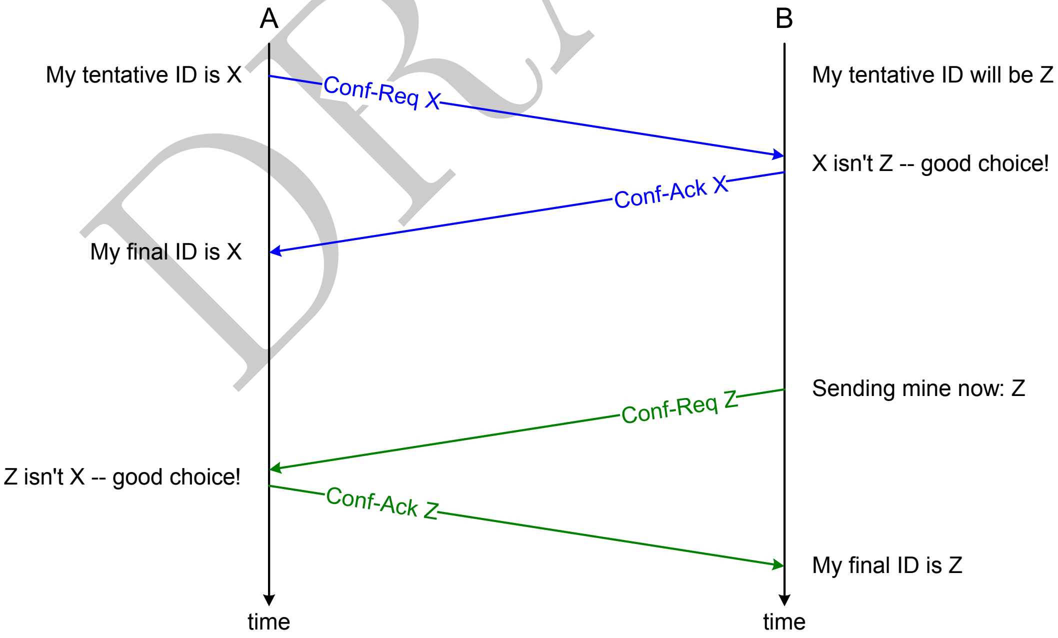


Figure 4.2: IPV6CP: consecutive negotiation without conflict

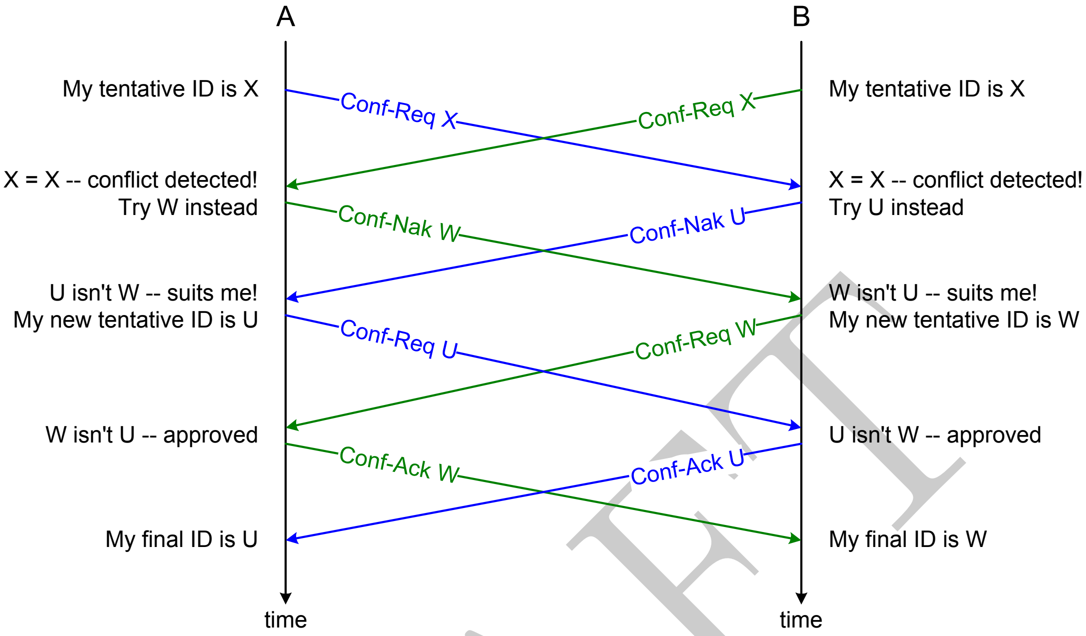


Figure 4.3: IPV6CP: concurrent negotiation with conflict resolution

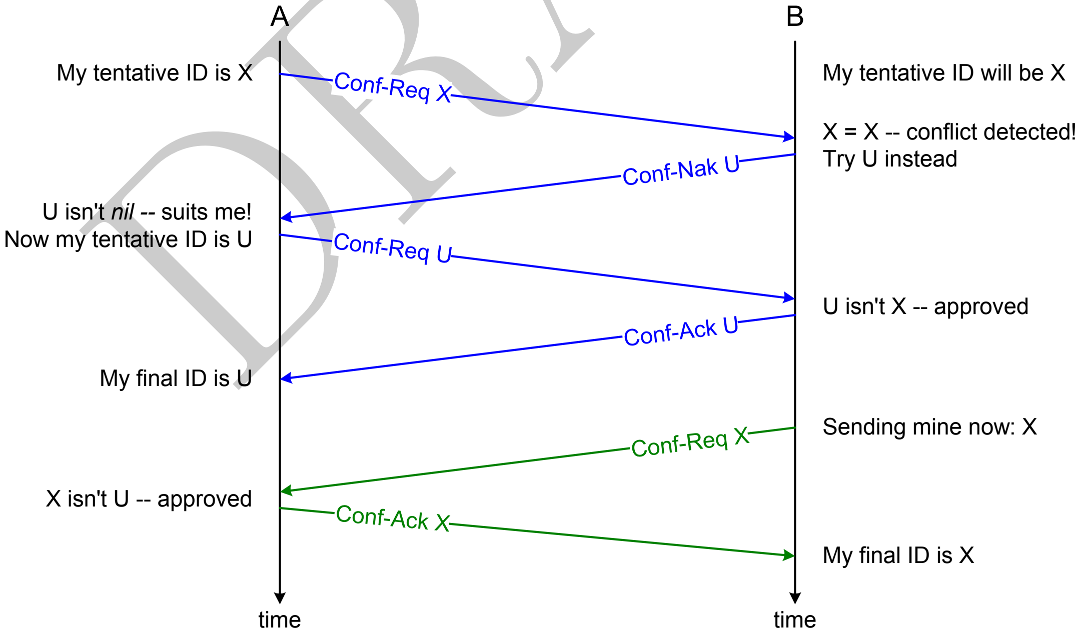


Figure 4.4: IPV6CP: consecutive negotiation with conflict resolution

In practice, an IPV6CP party may resort to harmless tricks in order to avoid a conflict in the very first round of negotiation. For instance, it can defer making its own choice of tentative ID until its peer sends a *Conf-Req* through. Once the remote peer's tentative ID is known, just a single bit can be flipped in it to get a different value. Drawing a conversation diagram for this scenario will be another exercise for the reader. Of course, the issue with this trick is which peer is supposed to send its *Conf-Req* first and how to avoid a protocol deadlock when neither of them steps up to do so. Although, from what can be seen on the wire, at least *Cisco IOS* 12.4 may play this trick when establishing a PPP link, a concurrent exchange of *Conf-Req* messages can be more reliable as it is free from the who-should-speak-first uncertainty.

4.2 Interaction with Upper-Layer Protocols

In the TCP/IP stack, there are several protocol classes that are the IP layer's direct consumers: control protocols such as ICMP; transport protocols as represented by TCP, UDP, and SCTP; tunnel protocols, for example, IP-IP, GRE, and EtherIP. The issue of ICMP will be dealt with in Section 4.3, and now let's have a general look at how protocols will be encapsulated in IPv6 followed by a little more focus on transport protocols.

Contrary to what some neophytes believe, IPv6 introduces no new transport protocols and so the good old TCP and UDP together with the marvelous young SCTP remain as relevant in the IPv6 era as they used to be in the IPv4 times. The point is, IPv6 is about modernizing the IP layer while keeping its interface for upper-layer protocols essentially the same.

The first thing to specify here is the encapsulation format governing how an upper-layer PDU is to appear within an IPv6 packet. In fact, this was already done in Section 3.1, while at the issue of IPv6 packet headers. As was said, every IPv6 packet starts with one or more headers chained through their **Next Header** fields and followed by actual payload, the latter normally being a PDU in its own right, such as a TCP segment or a UDP datagram. Thanks to **Next Header** values coming from the IP protocol registry [83], the last header in the chain can refer to the payload type in the same manner as the IPv4 header did, mutatis mutandis. For example, if a TCP segment is found in an IPv6 packet, its last Next Header value will be 6; if a UDP

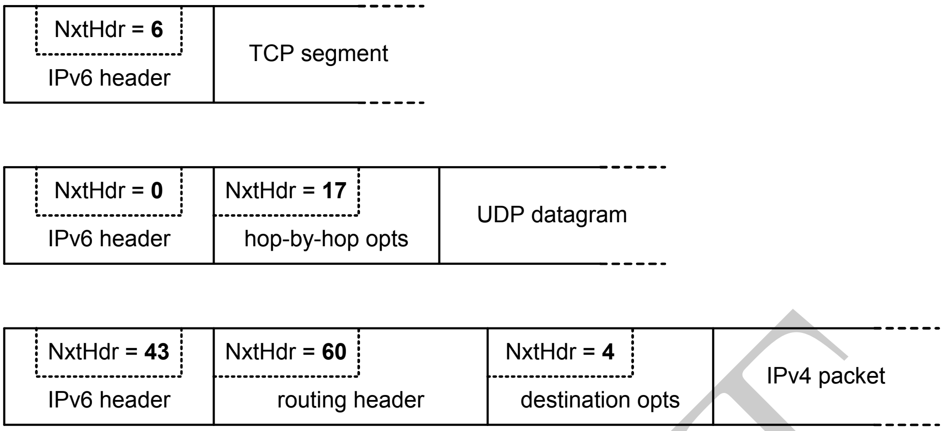


Figure 4.5: IPv6 encapsulation examples

datagram is encapsulated, that value will be 17; and if an IPv4 packet is tunneled, a value of 4 will be there—see Fig. 4.5.

The next action item for us will be to spot, and deal with, all obvious as well as hidden dependencies on IPv4 the upper-layer protocols can have. Fortunately, well-known protocols in active use don't have too many of those.

> By the way, there was an attempt to compile a comprehensive list of IPv4 dependencies in RFC protocols, the report coming in as many as eight (!) consecutive RFCs: 3789–3796.

The main dependency TCP and UDP have on IPv4 is the pseudo-header format used to checksum their PDUs. The original pseudo-header format included the IPv4 Source and Destination Addresses, and so a new format will be required for IPv6 environment [82, Section 8.1] as shown in Fig. 4.6.

> The reader will research whether pseudo-header considerations apply to SCTP. (Hint: *Use [126] for an introduction to SCTP.*)

The pseudo-header retains a length field to store the upper-layer PDU length in bytes, excluding any IPv6 headers. For example, UDP can just copy this value from the UDP header while TCP, carrying no explicit segment length value, will have to calculate it.

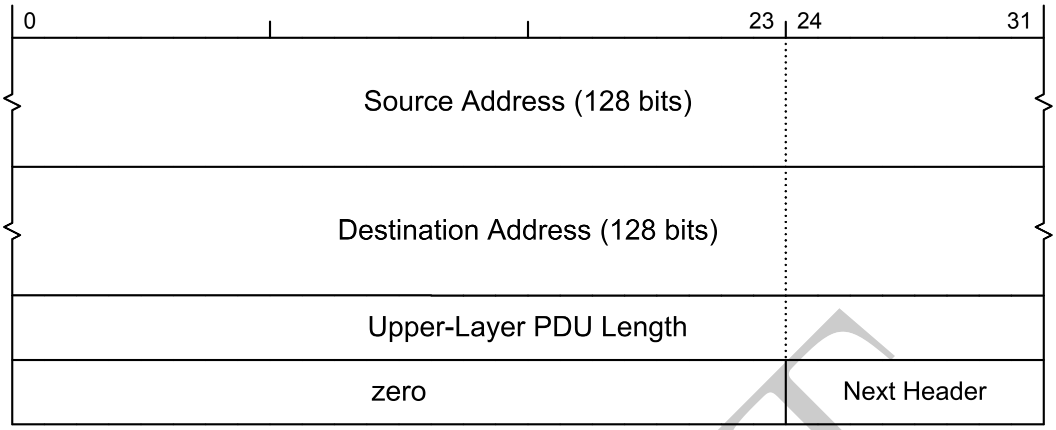


Figure 4.6: IPv6 pseudo-header for checksumming purposes

A straightforward calculation can subtract the total length of IPv6 *extension* headers present in the packet—not counting the main IPv6 header!—from the **Payload Length** value in its main IPv6 header. Alternatively, a C language adept would suggest finding the difference between the pointer to the end of the packet in memory and that to the beginning of the TCP segment enclosed, both cast to *char **. A tricky point will be not to confuse the end of the packet with its last byte—an archetypal off-by-one error.

As for the **Next Header** field, in the IPv6 pseudo-header it will need to contain the upper-layer protocol code, which won't always be the value from the main IPv6 header due to possible presence of IPv6 extension headers between the main IPv6 header and the upper-layer PDU.

Apart from the pseudo-header based checksumming, UDP implicitly relied on the IPv4 header checksum in that the UDP checksum was optional: It was assumed that IP could protect its own header integrity. The IPv6 header having no such protection by design, *UDP over IPv6 must checksum all its datagrams* [82, Section 8.1]. Should a UDP datagram with a zero checksum value arrive in an IPv6 packet, it must be discarded.

Let's refresh our memory on Internet checksum math [127, 128]. It is computed using 16-bit ones-complement arithmetic, whose most striking feature is in having as many as two binary values behaving as an arithmetic zero: `0x0000` and `0xFFFF`. (This peculiarity can lead to interesting pitfalls when trying to optimize calculations [129].) The final checksum value as stored in a TCP or UDP header is the sum of data words with its bits flipped, which means ones-complement negation. (No surprise that the sum of all words including the checksum field will be zero in a valid packet.) An extra rule therefore can allow to distinguish a genuine zero checksum from an undefined checksum: Should a UDP datagram have a genuine checksum of zero, it shall be stored only as `0xFFFF`, thus reserving the binary value of `0x0000` to indicate an undefined checksum. Do you find this trick simple and elegant?

NAT can be generalized to handle both IP versions and to enable "direct" communication between IPv4 and IPv6 hosts. Still, what can a NAT46 or NAT64 middlebox do if a non-checksummed UDP datagram is received on the IPv4 side? To forward it on to the IPv6 side, the datagram will have to be checksummed assuming its data is intact. Thus the middlebox will effectively certify the datagram's integrity without sufficient proof as it might have been corrupted on its way to the middlebox. This is yet another of the many ways in which NAT can violate the end-to-end principle.

And what effect will have the rise of IPv6 on tunnel protocols? Providing a tunnel protocol was designed right, it will require no changes to support IPv6. Take GRE for example: It will only need to take into account that IPv6 has an Ether Type value distinct from that of IPv4. As an extra, the optional GRE checksum can be enabled when tunneling IPv6 packets so as to ensure their integrity.

The most basic IP-IP protocol is as ready to handle IPv6 both outside the tunnel [130] and inside it [131, Section 3][132, Section 3.1]. All it takes is the knowledge that an inner IPv4 packet is indicated by the IP protocol value of 4 and an inner IPv6 packet, by the IP protocol value of 41 [83]. Furthermore, mixed-version tunnels are perfectly possible: IPv4-in-IPv6 and IPv6-in-IPv4. Their encapsulation rules are summarized in Table 4.2.

Besides, all tunnels need to be concerned with fragmentation management because tunnel encapsulation can reduce the effective MTU from what the underlying physical links provide. If the outer protocol is IPv4, then the ingress node, aka the encapsulator, can allow or prohibit the fragmentation of a tunnel packet by setting its **DF** flag appropriately. In contrast to that,

Table 4.2: Multi-version IP-IP encapsulation

Header(s)	IPv6-in-IPv6	IPv6-in-IPv4	IPv4-in-IPv6
Outer	IPv6: Next Header = 41	IPv4: Protocol = 41	IPv6: Next Header = 4
Inner	IPv6	IPv6	IPv4

IPv6 provides no transit packet fragmentation, affecting tunnels over it and complicating the encapsulator's job. In the latter case, it is up to the encapsulator to control the length of its tunnel packets so that they reach the egress node (decapsulator) rather than exceed an MTU and get dropped.

The tricky part here is that the inner MTU of an IPv6-compliant tunnel must not be less than 1280 bytes, the tunnel striving to be a link too, while there can be physical links in the tunnel's path whose MTU is 1280 bytes (or just slightly larger). IPv6 encapsulation of a 1280-byte inner packet will result in a tunnel packet never smaller than 1320 bytes and so having no chance to reach the far end of the tunnel unless fragmented beforehand. This means that an IPv6-ready tunnel encapsulator has to be prepared to fragment its packets because their fragmentation can't be avoided for good. The simplest, if inefficient, way to control the length of tunnel packets is to assume a PMTU of 1280 bytes and fragment tunnel packets to that limit. Then the inner MTU of the tunnel will be restricted only by the range of unfragmented IPv6 packet length less the tunnel overhead: $65575 - X$. A more elaborate approach will be for the encapsulator to do proper PMTUD.

A comprehensive review of the theory and practice of tunnel fragmentation management can be found in [133].

We generally refer to any underlying link whose nature is irrelevant at our current abstraction level as a "physical" link. It can equally well be a genuine CSMA/CD *Ethernet*, a VLAN, a VPLS, or even another IP tunnel. It is thanks to the fundamental concepts of stackable protocols and overlay networks that such abstraction can be afforded.

4.3 Control

If we really wanted to, we could probably try adapting the current ICMP as defined in [134] to serve IPv6 as well. However, we can't miss the opportunity

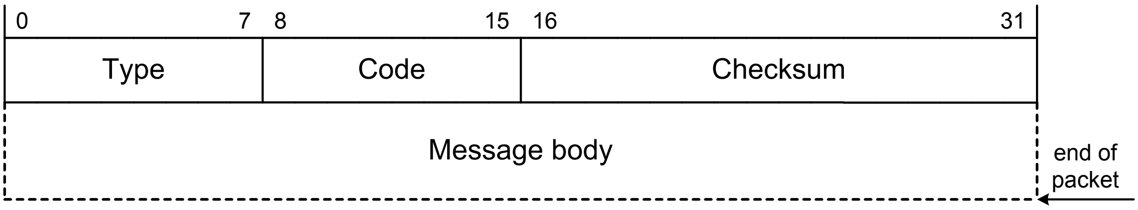


Figure 4.7: Basic ICMPv6 message format

to revise, along with IP itself, its control protocol. To avoid worries about compatibility with ICMP for IPv4, the new control protocol intended for IPv6 will get a distinct code from the IP protocol registry: 58. The newborn protocol's name will be "ICMP for IPv6" or **ICMPv6** for short [135] though neither ICMP header contains a version field. To eliminate ambiguity, the old ICMP will be referred to as ICMPv4 herein, at least when discussed along with ICMPv6.

> A necessary assumption is that an ICMPv6 message will never appear in an IPv4 packet and vice versa simply because neither combination makes sense. Ideally, an IP stack implementation should enforce it on incoming packets. In special cases such as NAT from IPv6 to IPv4 and back, control messages will need to be translated as well for the sake of completeness. The translation procedure and the very possibility of such translation between the ICMP versions are outside the scope of this book.

What will the requirements be for ICMPv6? First of all, it must be as flexible and extensible as IPv6. We will address this requirement by composing the basic ICMPv6 message format of two parts, one of them fixed and the other variable, as shown in Fig. 4.7. In fact, we have hardly invented anything new here: ICMPv4 messages had essentially the same common format.

The **Type** field value will be what indicates the meaning of the rest of the ICMPv6 message. For example, *Destination Unreachable* and *Echo Request* convey completely different information and so they clearly deserve different **Type** values. Furthermore, it will be **Type** that defines the actual format of the flexible part of the message.

ICMPv6 being a separate protocol, its **Type** values bear no connection with those of ICMPv4. ICMPv6 message types are going to be allocated from scratch and so there is an opportunity to organize them, if you will, by their "mood". In ICMPv4, error message types were intermixed with purely informational types and so it was hard to tell the message "mood" by the **Type** value alone. Now it will be handy to separate those classes of messages, e.g., as follows: Error message types will stay in the lower half of the range, that is, 0–127, and informational types will be confined to the upper half, 128–255. Then the "mood" of an ICMPv6 message will be indicated just by its type's high bit: 0 will flag an error notice while 1 will mean mere information [135, Section 2.1].

The interpretation of the **Code** value will depend on the message type. For example, **Code** can be used to subdivide the type into more specific groups, and we will make use of that in this section.

The format of the variable part to be found after the fixed part is as dependent on the ICMPv6 message type. That part will be known as the **message body**. It extends all the way down to the end of the enclosing packet, and so its length can be calculated from IPv6 **Payload Length** by the destination.

By the data integrity protection model adopted by IPv6, the ICMPv6 checksum scope is to include the most essential fields of the main IPv6 header in addition to the ICMPv6 message itself. Utilizing a pseudo-header of the standard format from Fig. 4.6 in Section 4.2 is the way to go here. The checksumming procedure using the pseudo-header remains the same, too: For checksum calculation, the pseudo-header is assumed to precede the ICMPv6 header and the **Checksum** field is initialized with zero [135, Section 2.3].

What will the value of **Next Header** need to be in the pseudo-header? (Hint: 58*, that is, ICMPv6.*)

Recall that ICMPv4 used no pseudo-header and its checksum scope spanned only the ICMPv4 PDU, with no IPv4 fields included. Thus the use of a pseudo-header by ICMPv6 is a new feature that has to be introduced due to no IPv6 header checksum.

When in an IPv6 packet, an ICMPv6 message is supposed to occupy all of its upper-layer payload section, thus making an ICMPv6 length field redundant: The length of the ICMPv6 message can be calculated by deducting the lengths of the IPv6 extension headers present in the packet from the IPv6 **Payload Length**. This calculation will be required at the destination to fill the pseudo-header out—consult Fig. 4.6 to see why.

Table 4.3: Essential ICMPv6 message types

Type	Name	Reference
	Error messages	
1	*Destination Unreachable*	[135, Section 3.1]
2	*Packet Too Big*	[135, Section 3.2]
3	*Time Exceeded*	[135, Section 3.3]
4	*Parameter Problem*	[135, Section 3.4]
100	Private experimentation	[135, Section 2.1]
101	Private experimentation	[135, Section 2.1]
127	Reserved for expansion of error messages	[135, Section 2.1]
	Informational messages	
128	*Echo Request*	[135, Section 4.1]
129	*Echo Reply*	[135, Section 4.2]
200	Private experimentation	[135, Section 2.1]
201	Private experimentation	[135, Section 2.1]
255	Reserved for expansion of informational messages	[135, Section 2.1]

> Keep in mind that “payload” can be ambiguous in IPv6. In the conventional sense, it is the last section of a packet that follows the main IPv6 header and extension headers, if any, and contains an upper-layer PDU such as a UDP datagram, a TCP segment, or an ICMPv6 message. In the specific sense of IPv6 **Payload Length**, it is the packet except its main IPv6 header. This issue is of particular concern to the final destination because, in order to validate the checksum where a pseudo-header is applicable, *both* meanings will effectively come into play: Specified in the IPv6 header is **Payload Length** but the pseudo-header needs to be filled in with the upper-layer PDU length.

The next issue for us to deal with is the assignment of ICMPv6 message types. Only the most essential of them will be assigned immediately [135, Section 2.1] as shown in Table 4.3, and for sundry other types, to appear later on, there will be a registry [136].

> One possible reason for Type 0 to stay unassigned could be that registered zero values are disfavored by some protocol designers due to the other use of zero as a placeholder meaning “no data”.

What changes from the essential ICMPv4 types can be pointed out? First of all, the *Packet Too Big* and *Time Exceeded* messages now have distinct types. At the same time, the *Destination Unreachable* message became more specific, indicating only a failure to deliver a packet caused by an addressing or routing problem, or by a policy restriction [135, Section 3.1]; for example:

- Code 0: No route to destination; that is, the packet couldn't be forwarded because the next hop was unknown.
- Code 1: Communication with destination administratively prohibited; e.g., blocked by a firewall rule.
- Code 2: Beyond scope of source address; meaning a zone violation was prevented, such as trying to send a packet to a destination out there on the Internet from a link-local source address.
- Other *Destination Unreachable* codes are listed in the ICMPv6 RFC document [135, Section 3.1] and the ICMPv6 parameter registry [136].

What is the benefit from *Packet Too Big* having a separate type instead of just a code within the *Destination Unreachable* type like in ICMPv4? First and foremost, that will help network security engineers to design IPv6 firewall rulesets that block random *Destination Unreachable* messages without breaking PMTUD. As all of us know too well, it was an issue in IPv4. On the one hand, a failure to allow this particular ICMPv4 type-code tuple was a common oversight on security engineers' part. On the other hand, a number of IPv4 firewall products on the market could match ICMPv4 types but not ICMPv4 codes, and so their operators had a hard time trying to allow PMTUD selectively. These considerations can't just be ignored in IPv6, where the importance of PMTUD has increased tenfold due to the new fragmentation rules where the packet length is controlled exclusively by the source—see Section 3.3.4 to refresh your memory on IPv6 fragmentation if necessary.

In a perfect world, ICMP error filtering shouldn't be an issue to a stateful firewall that can link ICMP error messages with application sessions tracked. All such a firewall would need to do is allow ICMP error messages proven related to a permitted session and block all stray ones. Unfortunately, not all firewall products handle ICMP signaling in its proper session context, but this is just a software deficiency to be eliminated in some future release. A more fundamental problem is that an ICMPv4 error message may be impossible to relate to a session based on what data is included in it. We will have to sort this issue out for IPv6 while at ICMPv6.

Next, an MTU field is a must in the IPv6 *Packet Too Big* message format so that the limiting MTU is signaled back to the oversize packet's source for the sake of a faster convergence of PMTUD—we came to this conclusion in Section 3.3.4.

The *Parameter Problem* message, Type 4, is there to notify the source about unsupported options, unrecognized **Next Header** values, and other possible problems in the packet it originated. What the problem was can be indicated by the **Code** value as follows [135, Section 3.4]:

- Code 0: Erroneous header field encountered. This is the default code that can mean a wider problem with the packet than just an invalid value of an isolated field. For example, it will be returned if the length of a fragment with $M = 1$ (more fragments) isn't a multiple of 8.
- Code 1: Unrecognized Next Header type encountered. The destination apparently had no support for this extension header or upper-layer protocol.
- Code 2: Unrecognized IPv6 option encountered. This message is to be returned only if the high bits of the option type require that, as provided for in Section 3.3.2.

But hang on. How did *Time Exceeded* creep back in as the name for Type 3? Was it not us who made the decision in Section 3.2 to acknowledge the modern semantics of the former **TTL** field in its new name, **Hop Limit**, stripping it of any reference to time? Don't worry; we aren't taking back what we said earlier. It is just that Type 3 is to accommodate two different event codes, the other one still time-related [135, Section 3.3]:

- Code 0: Hop limit exceeded in transit; that is, there probably was a routing loop in the network and the packet's **Hop Limit** dropped to zero.

- Code 1: Fragment reassembly time exceeded; that is, the packet was fragmented and the destination timed out waiting for all fragments to arrive.

> Unfortunately, the text of [135, Section 3.3] can be misunderstood as though the router must check **Hop Limit** in an incoming packet first thing after it was received from the wire. We know from Section 3.2 that this check may be performed only after making sure the packet is to be forwarded on, otherwise **Hop Limit** is to be disregarded. The authors of [135, Section 3.3] must have assumed the model of an ideal router, to which all packets are transit ones, but forgot to emphasize their assumption. Alas, today such implicit assumptions often give rise to non-conformant implementations.

Other ICMPv6 types will be introduced when needed. While their individual formats will certainly be tailored to suit their roles, there can still be common features in them. One such feature of importance is to be found in all ICMPv6 error messages.

The error signaling mechanism of IPv6 is essentially the same as we knew it in IPv4: *An ICMP error message is triggered by a failure to process an IP packet received from the network.* Consequently, each ICMP error message can be traced to a certain offending packet. Therefore it will be a good idea to include at least the beginning of the offending packet in the error message so that the offending packet's source can recognize the packet and act accordingly. This rule applies to all ICMPv6 error messages as much as it used to apply to their ICMPv4 cousins.

Is there anything to improve in the error signaling procedure before it gets adopted for IPv6? One such detail is how many bytes of the offending packet need to be included in the error message. In ICMPv4, the basic requirement was just the IPv4 header and 8 bytes of the payload [51, Section 3.2.2]. That proved insufficient to identify the original packet if the network path involved multi-level encapsulation such as tunnels.

Now we simply must use the opportunity given and require that *each ICMPv6 error message include as much of the offending packet as possible* [135, Section 2.4(c)], as shown in Fig. 4.8. Quite naturally, if the whole offending packet doesn't fit in, its start must be included rather than some arbitrary portion cut from its middle or tail [135, Section 2.1] because it is the start that provides the key to the packet contents.

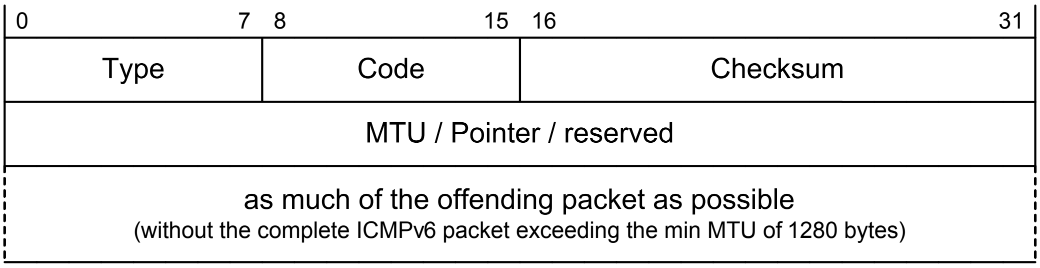

0
7
8
15
16
31
Type
Code
Checksum
MTU / Pointer / reserved
as much of the offending packet as possible
(without the complete ICMPv6 packet exceeding the min MTU of 1280 bytes)

Figure 4.8: Generic ICMPv6 error message

Discuss what can be included as the offending packet's copy when its reassembly timed out but the first fragment never arrived. (Hint: *One option is to include an Unfragmentable Part sample along with the Fragment Header.*)

Where can a cap on the packet sample length stem from, if at all? As outlined in Section 3.3.4, ICMPv6 error messages will take part in IPv6 fragmentation management through their involvement in PMTUD. Should they be subject to fragmentation themselves, our scheme will loop. The fragmentation of ICMPv6 error messages therefore must be avoided a priori, through keeping their length within the minimal MTU of 1280 bytes. Of course, the 1280-byte limit is to be met by the complete ICMPv6 packet including any IPv6 headers in front of its ICMPv6 header.

If we dig a bit deeper, it won't be hard to notice that the combination of these conditions (as much as possible but not more than. . .) rules out a number of cases, leaving room for just two of them:

- The error message contains a complete copy of the offending packet as far as available.

- The error message contains just the beginning of the offending packet and its IPv6 total length is 1280 bytes.

The newly acquired comprehensiveness and specificity of ICMPv6 comes at a price though: Now ICMPv6 error messages can be large and the resulting overhead will be significant if their rate is high, e.g., due to an attack leveraging the ICMPv6 signaling aspect of IPv6. This issue is further exacerbated by the indirect attack vector where the attacker sends a stream of invalid packets to node A, spoofing their source as node B and prompting the bombardment of node B by node A with large ICMPv6 messages. This problem can be precluded only if all IPv6 nodes rate-limit their ICMPv6 responses. Let ICMPv6 rate-limiting be mandatory in all IPv6 implementations [135, Section 2.4(f)] while its algorithm can be implementation-specific as long as it does the job.

> One such algorithm is outlined in [135, Section 2.4(f)].

A related issue for us to move on to is as follows [135, Section 2.4(e)]: What are the cases where an IPv6 node must refrain from sending an ICMPv6 error message altogether? First of all, to avoid a dangerous ping-pong, *an error message is never to be triggered by another error message.* Besides, there can be ICMPv6 informational messages prone to such a ping-pong, so we should keep a lookout for this aspect.

> For example, in Section 5.2 we will encounter an ICMPv6 message type known as Redirect, which is odd in that it can be triggered by an arbitrary IPv6 packet although it isn't an error message.

Yet another consideration is that an ICMPv6 error message must not be sent in response to a packet whose source address wasn't unicast. The rationale is that *the Destination Address of an ICMPv6 error message is the same as the Source Address of the offending packet.* For example, if the offending packet came from a multicast source address, it is a fault at best, or a DoS attack at worst, because the Source Address must not identify more than one interface—see Section 3.2. Furthermore, there are a few valid cases—to be discussed in Section 5.4 and Section 6.4—where the IPv6 Source Address can be left unspecified (*::*). As far as error signaling is concerned, packets sent from *::* shouldn't be reacted to either so as not to send an ICMPv6 error message *to* the unspecified address.

Lastly, multicast or broadcast packets must not be responded to with ICMPv6 by default simply because it is too easy for just one such packet to precipitate a whole storm of error messages. To detect a multicast or broadcast packet, its link-level header needs to be consulted in addition to its IPv6 header because it can be disguised with a unicast IPv6 Destination Address. For example, if the packet was received from an *Ethernet* link, its MAC Destination Address should be verified as well. So this check is possible only as long as the underlying link layer lends IPv6 a hand.

> For instance, a link-layer driver in the *BSD* network stack is expected to mark incoming multicast and broadcast packets with the *M_MCAST* and *M_BCAST* bit flags, respectively [137, p. 39], and so the *BSD* IP module is spared the gory details of link-level encapsulation and addressing.

There are just a couple of exceptions from the rule of no ICMPv6 response to multicast or broadcast [135, Section 2.4(e)]. First, if an unsupported IPv6 option has the most significant two bits of its **Option Type** set to `10`, an ICMPv6 *Parameter Problem* message is to be sent back regardless of the MAC and IPv6 Destination Addresses in the offending packet—see Section 3.3.2. Second, the ICMPv6 *Packet Too Big* message is always exempt from the destination address check [135, Section 3.2] so that PMTUD can be used on multicast as well as on unicast.

When an ICMPv6 error message is returned by a router, its Destination Address will (naturally) be the same as the Source Address of the offending packet; and what will its Source Address be? This is a non-trivial question [135, Section 2.2(b)] that can be answered only through using a complex procedure known as Default Address Selection—to be discussed much later, in Section 6.6. The biggest challenge here is to pick a local address whose zone is suitable for communication with the Destination Address. This issue wasn't in IPv4, where the router would just use one of its addresses on the interface over which the ICMP message was to be transmitted [84, Section 4.3.2.4]. Also compare this with the much simpler case of an IPv6 host signaling back a broken unicast packet that was properly addressed to the host nevertheless. In the host case, the Source and Destination Addresses just need to be swapped [135, Section 2.2(a)].

Chapter 5

Neighbor Discovery Protocol

5.1 Neighbor Discovery and IPv6 Address Resolution

According to our model of computer networks, packets aren't magically flown from one network node to another: They travel through certain links. Since we are concerned with the IP layer, our viewpoint is that direct transmission of a packet to another node is possible only as long as the other node shares a link with our local one.

> To help our thinking about communication between network nodes, we often resort to the favorite mental trick of protocol designers: to put oneself in a particular node's place and refer to it as the local node and to the others as remote nodes.

> The link-level transmission itself can be a process of arbitrary complexity, even one recursively invoking the IP layer as done by tunnels. Only thanks to deliberate and careful separation of stack layers in the network architecture can such a transmission be completely transparent for the consumer layer and appear to it as an atomic event.

In order to avoid coming to wrong conclusions and having to overturn them afterwards, we need to realize early on that having the nodes connected to one link is required but not quite sufficient to make direct transmission possible between them. The issue is, there are unconventional links where not all nodes can speak to each other. Section 5.2 will be dedicated to a detailed discussion of such odd links in IPv6 context, and for now let's assume that our local node *A* has a way to know it can transmit a packet to

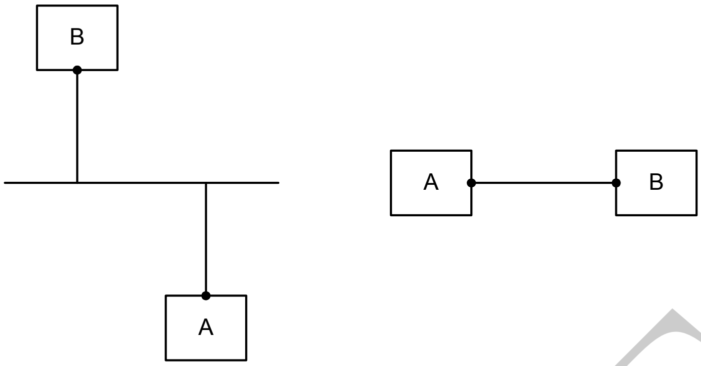


Figure 5.1: Neighbor nodes on a broadcast link (left) and on a point-to-point link (right)

node B using some link L. If that is the case, nodes A and B are said to be **neighbors** on link L. Until we get too sophisticated through the "forbidden knowledge" of unconventional links, we can indulge in a nice familiar picture where neighbors A and B are connected with a good old PPP line or a trusty *Ethernet* LAN, as illustrated in Fig. 5.1.

This initial simplification notwithstanding, let's keep in mind from the outset that neighbor relation, as a mathematician would put it, enjoys commutativity but lacks transitivity: If node A is a neighbor of node B, then node B is a neighbor of node A; however, if A and B are neighbors and B and C are such, that doesn't automatically mean that A and C also are.

Technically, neighbor relation is commutative only as long as the link is bidirectional. We need the notion of neighbors only inasmuch as it can help us to develop certain mechanisms for bidirectional links, so we are free to disregard one-way links here.

Another property of neighbor relation is its reflexivity, whereby any node is its own neighbor and needs no router to speak to itself. However natural may it seem, it shouldn't be taken for granted because it is implemented using a loopback interface and a virtual one-node link connected to it, which is where this degenerate case of neighborship materializes. Whenever this scheme fails, artifacts appear, such as a PPP node speaking to itself via the far-end peer.

The next question is how node A is going to call its neighbor B. Will it do that by nickname, title, or call sign? Hardly so. Since we are concerned with TCP/IP, node A will need to use an IP address assigned to B's network interface into the shared link L.[1]

In particular, when a routing decision has been made on an outgoing unicast packet, node A knows only the IP address of the remote neighbor to hand the packet off to. Because that neighbor can be a router or a destination, it is referred to as the next hop, implying it may not be the last step in the packet's path through the network.

> Sometimes, the next hop off the host originating the packet is called the first hop. This fine distinction is seldom important, the first hop being just a special case of the more general next hop.

However, in order to actually get the packet out, the IP layer needs to invoke the link layer, which knows nothing of IP addresses and needs the link-layer address of the next hop neighbor given in an explicit or implicit form. The local node therefore has to find out first which of its neighbors on link L possesses the IP next hop address, and to learn the link-layer address of that neighbor. This procedure is known to us as address resolution.

If the link is point-to-point, address resolution is a trivial task because the explicit IP address and the implicit link-layer address of the far-end neighbor are known beforehand.

> Let's recall that implicit addressing can be sufficient in certain cases. For instance, any form of explicit link-level addressing would be redundant on a point-to-point link because a transmission over such a link will be received by a single node, the one it was intended for. One can say that the link-layer address of the neighbor on a point-to-point link is spelled, "the node over there at the far end of the line". Consequently, using MAC addresses on a point-to-point link would be redundant although it isn't entirely prohibited.

On the other hand, if the link is a multi-access one, the problem becomes complex, its solution depending heavily on the fine properties of the link. At worst, with no auxiliary mechanisms available, the only viable solution will be a static map from IP addresses to link-layer addresses populated by hand. But, fortunately, such minimalist links are now extinct while real-world links usually offer a means to address all neighbors at once and ask who owns the IP address we are after, as illustrated by Fig. 5.2. So automatic **neighbor discovery** [139] is made possible.

[1] Node B can have more than one such interface.

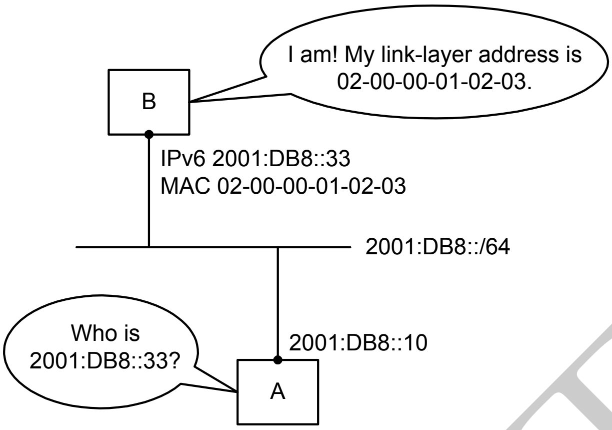


Figure 5.2: Automatic neighbor discovery: the concept

> The option to address all neighbors at once is offered not only by broadcast links such as *Ethernet*. For example, any point-to-point link also provides support for broadcasting and multicasting because there is just one remote neighbor on it to speak to. As for a true broadcast link, its actual guarantees are even stronger than that: All its nodes are neighbors with each other and, in addition, each node can broadcast to the rest of the bunch.

Although address resolution and neighbor discovery may at first seem just the same thing, in fact their scopes only have some overlap. In general, neighbor discovery can be performed using any other criteria besides the IP address. For instance, the local node could be asking its neighbors, “Hey there, who of you guys is the default router?” or, “Say, whose is that whiny fan?” On the other hand, address resolution doesn’t always require neighbor discovery. For example, in Section 4.1.1 was discussed how, and why, the resolution of IPv4 and IPv6 multicast addresses to MAC multicast addresses is 100% local, boiling down to a simple bitwise operation. Neither is network-based discovery necessary when the info required is already available in the local node’s configuration—what we knew as static ARP in IPv4.

> Examples of address resolution done locally can also found in the history of computer networks. E.g., the interface ID in an IPX address was always the same as the interface’s MAC address, and so the latter could be found as easily as by trimming off the high-order bits of the IPX address that contained a subnet prefix.

We need to keep this logical distinction in mind so as not to come up with a limited and inflexible solution. That being said, our immediate task is IPv6 unicast address resolution based on neighbor discovery, where the local node A will ask, "Dear neighbors, which of you owns address B?"—and hear back the link-layer address of that neighbor provided it is actually present on the link.

> An impatient reader can find a comprehensive list of functions for the IPv6 Neighbor Discovery to perform in [139, Section 3]. There is no need to rush ahead though, for we are going to meet those functions in due course.

To avoid reinventing the wheel if possible, we will see first if the solution from our IPv4 toolkit can do the job: ARP. To give it its due, ARP was a nice flexible protocol able to work with different Layer 2 and Layer 3 address families, and now we would have no problem adapting it to IPv6. But there still is a valid reason to give up on ARP and design a new neighbor discovery mechanism for IPv6 from the ground up. What we are going to disfavor ARP for is its. . . reliance on broadcast. How so? Did we not just say that broadcast made neighbor discovery possible? No doubt, it was a nice starting point, but there still is a way to go from the raw idea to a good implementation. What we should do now is take modern link features into account. Back in the old days, unicast and broadcast were the only options available, but since then multicast has also been adopted by the majority of link-layer technologies.

What are the advantages of multicast over broadcast? First of all, broadcast has no selectivity soever. Suppose a link is used in a multi-protocol environment where some nodes support only IPv4, some do IPv6, some speak exclusively ISO, etc. Every time an IPv4 node broadcasts an ARP request over that link, the resulting broadcast frame has to be received, parsed, and only then discarded by the nodes that care nothing about IPv4. Thus each ARP request wastes the resources of non-IPv4 nodes and the link itself, and in a large network the total amount of "virtual energy" wasted so can be quite significant.

> An urban legend has it that some huge flat LANs are so overburdened with broadcast that only a most up-to-date, powerful computer can be connected to such a LAN without the risk of getting bogged down by broadcast traffic processing; but if your computer is a bit old and slowish, it will literally freeze trying to receive the endless stream of irrelevant broadcast frames only to discard them afterwards.

> By the way, the power use by the components of a network also depends on how well its traffic is optimized. This doesn't only cause environmental concerns: Battery life of smartphones and other handhelds can be severely reduced by too much stray traffic on a wireless LAN.

This situation could be much improved if neighbor discovery made use of multicast. In a simple bus-topology link, such as a classic *Ethernet*, unwanted multicast traffic can be filtered off by the network interface cards, and in a modern switched LAN such traffic won't even reach those segments where no node is interested in it. So the basic task for us will be to get IPv6 neighbor discovery adopt multicast.

> For example, a switched *Ethernet* LAN can optimize the propagation of multicast either by means of a specialized protocol such as GMRP [141] if supported, or through utilizing a trick known as "IGMP/MLD snooping" [142], to be discussed in Section 6.4.

IPv4 had a well-known multicast group encompassing all IPv4 nodes of this subnet, *224.0.0.1*, and that group's IPv4 address would map to a certain link-layer multicast address, e.g., the MAC-48 *01-00-5E-00-00-01*. However, it wasn't quite possible to base a *standard* mechanism of neighbor discovery on that because multicast support never gained mandatory status in IPv4 [51, Section 3.3.7].

Fortunately, times have changed and IPv6 multicast now is an essential feature of the IPv6 stack, as was decided in Section 2.8. Thanks to the scoped address architecture introduced in Section 2.4 and the IPv6 multicast address format from Section 2.8, a multicast address can be *constructed*, rather than just prescribed, to mean all IPv6 nodes of this particular link: *FF02::1*. If IPv6 neighbor discovery were based on it, that would instantly limit the nodes concerned to those actually running IPv6. Still, is this the best solution we can come up with?

The clue is in the impressive length of an IPv6 address. What if the multicast address to send the discovery request to includes the most characteristic bits of the unicast address looked for? Suppose a link connects a number of IPv6 interfaces in one subnet. Then, naturally, each interface will need a unique interface ID for its IPv6 address. In addition, an IPv6 interface ID is known to be 64 bits long. These qualities make it a first-approximation candidate we can try copying into the multicast destination address, to be combined with a well-known multicast prefix allocated for the purpose. Link-local scope will be appropriate for this multicast address.

Assuming for illustration purposes only that the well-known prefix is *FF02::/64*, the unicast address of *2001:DB8:1:2:3:4:5:6* (U) will make the multicast address of *FF02::3:4:5:6* (M). Now node B owning the unicast address U needs to have joined group M and be listening to neighbor discovery requests sent to that group. When its neighbor node A wanting to speak to address U sends its discovery request to group M, it will selectively hit node B simply because no other node on this link has an interface ID of `3:4:5:6`.

This may look like a clever plan on paper, but what will actually happen on the wire? First of all, in order to send the discovery request, node A will need to resolve the multicast destination address M to its link-layer counterpart. This operation will pose no difficulty because it is supposed to be completely local, as was postulated in Section 4.1.1, but some of the group ID bits will probably be lost. For instance, in the *Ethernet* encapsulation, only as few as 32 low-order bits of the IPv6 multicast address make it into the MAC-48 one. Besides, each group consumes its share of resources when in a smart LAN optimizing multicast traffic because all of the switches need to keep track of each group's "geography". Considering all this, we need to put a tighter limit on the number of multicast groups that can ever appear on a single link due to operation of IPv6 neighbor discovery.

One option here is just to copy fewer bits from the target unicast address U to the multicast destination address M. By our idea, they still need to come from the interface ID part of U. When interface IDs are allocated by a person, they are likely to be small numbers such as 1, 2, 3... On the other hand, when an interface ID is derived from a MAC-48 address by the procedure from Section 2.7, the MAC address format comes into play. As we know, 24 of its high-order bits are OUI. If the owner of the network stocks on NICs from just one vendor, the OUI in their MAC addresses will be the same or coming from a limited set of values. At the same time, 24 low-order bits of a MAC-48 address, forming its extension identifier, can be quite random because they need to be unique within one OUI and each vendor assigns them independently of its competitors. Finally, when the interface has a full-blown EUI-64, its extension identifier is 40 bits long but it still can be expected to have more randomness in its low-order bits. For example, if the vendor assigns serial extension identifiers to NICs manufactured, equipment from the same batch can end up having, say, the same top half in its extension identifiers but unique bottom halves. All in all 24 low-order bits of the interface ID can remain a good choice under different policies for interface ID and link-layer address assignment. Then the number of multicast groups IPv6 neighbor discovery can create on one link will never be over 2^{24}, which is still a huge number but nowhere near as vast as the original 2^{63}.

> Try to figure out from what we discussed earlier why 2^{63} not 2^{64}. (Hint: *Modified EUI-64 still has the **I/G**, aka **g**, bit in it, and neighbor discovery is concerned only with unicast address resolution.*)

Thus we have arrived at the reasonable decision that only 24 low-order bits of the target unicast address U are to be copied to the special-purpose multicast address M the discovery request will be sent to. Consequently, the well-known link-local multicast prefix to combine those 24 bits with needs to have 104 bits in it. The one actually assigned for this role is *FF02::1:FF00:0/104* [21, Section 2.7.1]. For instance, our sample address U, which was *2001:DB8:1:2:3:4:5:6*, will make this address M: *FF02::1:FF05:6.* The technical term for any link-local multicast address created to this recipe is **solicited-node multicast address**.

> The vocabulary of IPv6 neighbor discovery significantly different from what we were used to in ARP, we will try to show in due course where each of the new terms stemmed from. However, at times it has to go back to front, and here we have just learned that the node looked for is known now as the **solicited node**, also referred to by the more ARP-esque term **target node**. We are going to see shortly what soliciting has got to do with neighbor discovery.

Alright, now we know what address a neighbor discovery request about address U is to be sent to: It is the matching solicited-node multicast address M, which effectively singles out the interfaces on this link having an IPv6 unicast address with the same 24 low-order bits as those in the target address U. Even though the solicited-node multicast group can consist of more than one interface, through its use IPv6 neighbor discovery gains a considerably greater precision than if using broadcast or an All Nodes multicast destination. Of course, to support this mechanism, *an IPv6 node, when assigning a unicast address to any of its interfaces, must join the corresponding solicited-node multicast group on that interface.* Quite naturally, the opposite is also true: When an IPv6 unicast address is removed from an interface, the associated solicited-node multicast group must be left—unless there are more addresses on that interface having the same 24 low-order bits. Our first problem has been solved.

At first glance, an IPv6 node will have to join as many solicited-node multicast groups on an interface as the number of unicast addresses configured on that interface. However, we need to keep in mind the simple fact that *IPv6 unicast addresses having the same 24 low-order bits are mapped to the same solicited-node multicast address.* Thanks to that, the number of mandatory groups for an IPv6 node to join can be reduced through a careful choice of the interface identifiers in the node's IPv6 addresses. The simplest option is to stick to a single interface ID per interface such as the one based on the interface's own Modified EUI-64. In that case the interface's unicast addresses will differ only by their subnet prefixes, and their respective solicited-node multicast addresses will certainly be just the same. Why to make a point of keeping the number of such multicast groups in check? First, as was discussed in page 178, each multicast group costs a certain amount of network resources. Second, modern NICs can filter incoming multicast traffic by the MAC destination address to reduce the load on the bus and the CPU but the size of their programmable filter is usually limited. Should the number of groups listened to exceed the hardware filter size, the NIC will have to pass all multicast frames to the CPU in the hope that the CPU will be able to sort them out.

The solicited-node multicast address is just one possible use of IPv6 multicast for selective discovery of network resources. In general, it will go as follows: The resource owner node does a one-way transform from the resource name to a multicast address and joins that multicast group. The querier knowing the resource name performs the same computation to find the address to multicast its query to. For example, a node can be looked up by its symbolic name if a hash of that name in a certain text encoding is appended to a well-known multicast prefix. There is an experimental protocol designed along these lines to query IPv6 nodes for miscellaneous information [138].

The next problem for us to ponder over is what the IPv6 neighbor discovery protocol will look like. Even its stratum in the network stack is yet to be determined. However, its use of Layer 3 multicast inevitably places the new protocol on top of IPv6 rather than beside it, unlike IPv4 and ARP. In essence, neighbor discovery is an aspect of IPv6 control, so it won't be a bad idea to make IPv6 neighbor discovery a subset of ICMPv6. A protocol in its own right, this subset will be known as **Neighbor Discovery**, in title case to distinguish it from the universal concept of the same name, or **ND** for short [139].

> Consequently, ND messages will be encapsulated in IPv6 packets and so subject to packet filters and other network policy restrictions. Extra attention therefore will be required in order not to block legitimate ND traffic because such a mistake will break the address resolution mechanism and, through that, the transmission of all unicast traffic.

Positioned as it is in the network stack, ND will be able to operate over any IPv6-capable link. Granted, not all link types need neighbor discovery as such; for example, it will clearly be redundant on point-to-point links; but we are going to build other mechanisms based on ND that will be relevant to virtually all link types. Therefore we need to adopt the following principle from the beginning: *ND will support all IPv6 links with multicast capability.*

Does this stratification of ND mean that its messages will be routable and so able to cross link boundaries? Indeed, ICMPv6 messages can be as routable as any IPv6 packet. At the same time, the very definition of neighbor relation dictates that ND packets stay within the relevant link limits. The threat to the ND protocol is not so much in local ND messages leaked to other links as in malicious ND messages slipping in to the link from elsewhere.

Can the destination of an ND message detect if its real source wasn't a neighbor? The answer depends on whom the destination can trust. If it trusts no other node, the answer may be just negative unless some extra conditions are provided. On the other hand, if the destination can trust the routers of the link it is connected to, it can at least rely on those routers decrementing **Hop Limit** in packets they forward into the link. Alone this fact doesn't really help because the destination has no idea what the initial value of **Hop Limit** was. Can this value be just enforced? Suppose we required that **Hop Limit** be initialized to X in all ND packets. Then the final destination would compare the **Hop Limit** value it saw in the ND packet with X and, if they were equal, conclude that the ND packet actually came from a neighbor because it had never been forwarded by a router or intermediate destination.

However, an attacker can circumvent this protection against remotely injected ND packets if she finds out or guesses the number of hops H to the destination and, assuming $H = 1$ means neighbors, sets **Hop Limit** to $Y = X + H - 1$. How can this attack be countered? By using the same weapon of school arithmetic! In this scenario, $H > 1$ because the attacker isn't a neighbor—if she were, the destination would be in real trouble! Consequently, the attacker will have to increase the initial **Hop Limit** from X for a successful attack: $Y > X$. The only case where she will fail to is when X is already at its maximum, that is, 255.

> This trick can do its job only provided the routers do the right thing with respect to handling a zero **Hop Limit** as we discussed in Section 3.2. Otherwise decrementing it modulo 256 will yield 255.

> We are yet to consider ND-based attacks by neighbors.

The trick just discovered can thwart remote attacks not only against ND and not only in IPv6 environment since all it needs is a **Hop Limit** or **TTL** field in the packet header. For its broad applicability, this ingenious trick has been given a mildly pompous name: **Generalized TTL Security Mechanism**, **GTSM** for short [143].

> Another inherently routable protocol that in most cases has to avoid getting routed is BGP. Historically, the idea of TTL-based protection first appeared in the context of BGP [143, Section 4] and later it was generalized.

> It goes without saying that the use of GTSM can't be an excuse to neglect other security mechanisms such as packet filtering at the link perimeter. The fact is, effective long-term defense has to be defense in depth, i.e, that of a layered design.

Therefore *ND participants must use GTSM*: The sender of an ND packet must set its **Hop Limit** to 255 and each recipient must verify that the value hasn't been decremented from that [139, Sections 6.1, 7.1, 8.1].

Now we are free to move on to the issue of ND message types. To resolve an IPv6 unicast address, just two message types are sufficient, appearing at first glance as a request and a reply. However, even ARP, where exactly these message names were adopted, had to break a strict request-reply sequence for what was known as gratuitous ARP: a node announcing its link-layer address in case it might have changed.

> It was irrelevant whether the gratuitous ARP was a request or a reply because ARP receiver's algorithm [144] would check the opcode only *after* updating the local ARP cache.

Was it Confucius who said that using inaccurate names would make the words disobey? So we should fix up the terminology first. The message to carry the actual information on a node's IPv6 and link-layer addresses will be known as **Neighbor Advertisement** (**NA**). A node may transmit a Neighbor Advertisement voluntarily and is supposed to upon request. Other nodes can solicit such an advertisement when needed by issuing a **Neighbor Solicitation** (**NS**).

Equipped so, let's revisit the scenario where node A has an IPv6 unicast packet to pass to neighbor B but needs to learn B's link-layer address first, knowing only its IPv6 address. Now node A has got a straightforward procedure to follow:

1. Construct a solicited-node multicast address (M) by combining 24 low-order bits of B's unicast address with the well-known prefix *FF02::1:FF00:0/104*.
2. Build a Neighbor Solicitation message—its format has to remain open until we work out all fields required, but it will have materialized by the end of this section.
3. Multicast the Neighbor Solicitation to group M.
4. Wait for a Neighbor Advertisement from node B.
5. Extract B's link-layer address from the Neighbor Advertisement received.
6. Direct the queued packet in due link-layer framing to B's link-layer address.

And what will node A do afterwards? If it lets go of B's link-layer address as soon as the packet is out, it will have to repeat the address resolution procedure for each outgoing packet to B, resulting in a completely unacceptable overhead. On the other hand, node A shouldn't hold on indefinitely to the neighbor's link-layer address learned once because it may change in the future, for example, if node B has its NIC hot-swapped.

> Discuss what the most annoying problem would be if the address resolution procedure had to be executed once per outgoing packet. (Hint: *A delay equal to the link's RTT would be introduced.*)

From our experience, we can tell right away that a cache is necessary here. ARP had one, too, in order to keep the most recent mappings from a Layer 3 protocol address to a link-layer address. Its update procedure was quite simple [144]:

- A new entry was inserted on any ARP message from any neighbor as long as the message's **Protocol Target Address** was ours.
- An existing entry was updated on any ARP message from that neighbor, even if it was intended for someone else.
- An entry would expire on a fixed timeout.

Unless additional ARP messages came from the neighbor, its ARP cache entry would expire unconditionally, even disregarding that an upper-layer protocol such as TCP might still have communication going on with the neighbor, thus confirming the ARP cache entry's validity. To add insult to injury, an ARP request would be broadcast each time the live entry expired, wasting other neighbors' resources. The root cause of this suboptimal behavior was in a lack of feedback from the upper layers to ARP.

> The less than optimal way to manage the ARP cache was at least partly encouraged by the IPv4 Host Requirements RFC [51, Section 2.3.2.1]—see Item 1 therein. Notwithstanding the more efficient ARP cache management options offered along with it, Item 1 dominated implementations since the *BSD* network stack [137].

> Of course, from the theoretical perspective, only a host is supposed to have upper layers in its stack, such as the transport layer of SCTP, TCP, and UDP, whereas an ideal router can be concerned with nothing but forwarding transit IP packets, upper-layer aspects being of no interest to it.

We will see to it that ND inherits none of ARP's shortcomings. Still, the cache of IP-to-link-layer address mappings remains a really handy device for ND to adopt. In this context, it will be referred to as the **Neighbor Cache** (**NC**).

The first thing to tell about the Neighbor Cache is that there will be a feedback channel from the upper layers in the stack down to ND. This channel concerns the local host only and so its details can be left up to the implementation rather than prescribed by the standard. It can be as simple as a kernel function call. In any case, an upper-layer protocol driver will have a chance to signal down to the ND module that destination such-and-such is reachable OK. For example, the TCP will report a destination's reachability when new data or new acknowledgements (not duplicates) come from it.

It may seem that the TCP can't really infer the reachability of a destination from some packets coming from it because the NC entry can be confirmed only by a successful *transmission* to the neighbor as opposed to a reception from it. However, new TCP data and acknowledgements can keep coming in only as long as the remote peer is receiving our own acknowledgements and data, respectively, thus proving our own transmission successful.

In theory, it was possible to implement a similar feedback channel to ARP, too. Item 4 of [51, Section 2.3.2.1] provided approval for it, although it spoke of negative rather than positive signaling where the upper-layer protocol module would signal destination's unreachability down to the ARP for it to invalidate the corresponding cache entry.

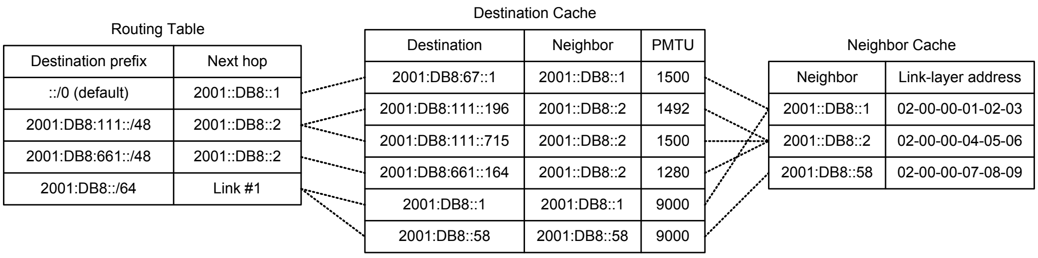
The fine details of the upper-layer protocol feedback mechanism for ND are discussed in [139, Appendix E.1].

But there is a link missing from our scheme: Upper-layer protocols are concerned with IP destinations and not the neighbors those destinations are reachable through whereas working out the next-hop neighbor, aka routing, is the IP layer's job. So we need to bridge the gap between the destination as reported by an upper-layer protocol and the neighbor corresponding to it.

Unlike forwarding, which applies to transit packets only, routing has to be done on all outgoing packets including those originated locally.

If the destination is just a neighbor, its NC entry can be refreshed immediately on a signal from an upper-layer protocol. But if it is reachable only via a router, the ND module will have to recall which next hop the last packet to that destination was sent through. The ND's "short-term memory" to keep and provide this information can be implemented as yet another cache, one storing the most recent destination-neighbor tuples. So its name will be the **Destination Cache** (**DC**).

A DC entry can also contain other useful info about this destination such as its current PMTU value. We will come to this idea naturally in Section 6.5.



Routing Table

Destination prefix	Next hop
::/0 (default)	2001::DB8::1
2001:DB8:111::/48	2001::DB8::2
2001:DB8:661::/48	2001::DB8::2
2001:DB8::/64	Link #1

Destination Cache

Destination	Neighbor	PMTU
2001:DB8:67::1	2001::DB8::1	1500
2001:DB8:111::196	2001::DB8::2	1492
2001:DB8:111::715	2001::DB8::2	1500
2001:DB8:661::164	2001::DB8::2	1280
2001:DB8::1	2001::DB8::1	9000
2001:DB8::58	2001::DB8::58	9000

Neighbor Cache

Neighbor	Link-layer address
2001::DB8::1	02-00-00-01-02-03
2001::DB8::2	02-00-00-04-05-06
2001:DB8::58	02-00-00-07-08-09

Figure 5.3: IPv6 node data structures and relations between them

Although a router's ND module can be getting no signals from upper-layer protocols at all because such protocols can be just left out from the router's stack, a Destination Cache can still be of use to the router. For example, when doing equal-cost multi-path (ECMP) load sharing, it is important to ensure that all packets of one flow such as a TCP session go via the same path; otherwise subtle problems can be introduced [145]. The one probably hit most often is packet reordering caused by the delay slightly varying across the paths. Of course, packet reordering isn't fatal to TCP/IP, but its constant presence can have a negative impact on the performance of transport protocols. E.g., the TCP congestion control mechanism will take an out-of-order packet for a sign that the previous packet was lost due to congestion [150, Section 3.2]. Another serious issue is apparent PMTU instability if PMTU through the paths isn't the same. A remedy to all these problems, the Destination Cache will help the router to forward all flows for one destination over the same path.

To get a better grip on how the newly introduced data structures are linked with each other, consider Fig. 5.3, which uses the familiar routing table as a starting point. Keep in mind though that in Section 5.2 we will revise the IPv6 reference host model and arrive at the somewhat surprising conclusion that the routing table is optional in it. On the other hand, an implementation can keep all information from the three conceptual structures in one place [139, Section 5.1] such as a radix tree built along the lines of the *BSD* routing table [137, p. 559].

More will be told about the Destination Cache in Section 5.2, and now let's focus on the structure and life cycle of a Neighbor Cache entry. Its function requires the neighbor's IPv6 address be its key; consequently no two NC entires can be about the same IPv6 address. On the other hand, Neighbor Discovery makes no attempt to detect whether different IPv6 addresses belong to the same neighbor interface or node. For this reason, NC entries are kept per address rather than per remote interface or node, and so by "the NC entry for neighbor B" we will mean the entry for address B. It is a basic assumption of the Neighbor Discovery model that a neighbor's identity is its IPv6 unicast address. While that may not be universally true, it works fine in the particular case of IP unicast address resolution.

> Later, when discussing IPv6 multihoming and multipathing in Section 6.8, we will encounter a fundamentally different situation where it is of the essence to keep track of which IPv6 addresses are owned by the remote node, but it will have little to do with Neighbor Discovery.

Initially the NC entry for neighbor B just doesn't exist. Its nonexistence lasts until our local node A has a packet to pass to next hop B.

> Note a difference from ARP here: An ARP cache entry would be created on any incoming ARP message whose Target Protocol Address was the local node's. Consequently, unnecessary entries could appear in the ARP cache. By contrast, an incoming Neighbor Advertisement can *never* trigger the creation of a new Neighbor Cache entry.

Then node A creates the entry but its link-layer address isn't known yet. Such an entry is said to be in *INCOMPLETE* state. To fill it in, node A needs a Neighbor Advertisement from B. It asks for one by sending a Neighbor Solicitation about the target address B to the solicited-node multicast address derived from the unicast address B. Once the NS message was sent, node A has just to wait. Should it wait for too long,[2] the NS will need to be repeated a few times,[3] since packets and frames are sometimes lost even in the most refined networks. Ultimately, if the NS retransmissions didn't help and nothing is heard back from B, the NC entry can only be deleted, the queued packet dropped, and its source notified via ICMPv6 if applicable.

> ICMPv6 Type 1 Code 3 is to be used in this case [135, Section 3.1]. Review Section 4.3 for when it won't be appropriate to send back an ICMPv6 error message.

[2] Parameter *RetransTimer*, default setting *RETRANS_TIMER* (1 second) [139, Section 6.3.2].

[3] Twice by default, making up to 3 solicitations in total—parameter *MAX_MULTICAST_SOLICIT* [139, Section 10].

Incomplete state for ARP cache entries was featured in the *BSD* IPv4 stack [137, Chapter 21]. It wasn't strictly necessary in ARP since the entry would be re-created even if the reply came as late as after the incomplete entry had timed out. The *BSD* stack used such entries for internal purposes, e.g., to keep track of queued packets waiting for address resolution to complete.

Now suppose a Neighbor Advertisement from B came in time. Does that mean that node B is mutually reachable and its link-layer address so learned can be used with confidence? Strictly speaking, no, since there is no proof yet that this NA was sent in response to an NS from node A. It can be that there is a one-way fault in the link and no NS from A ever reached node B but B sent a gratuitous NA for some reason, which would be perfectly OK. To distinguish these two cases reliably, the NA format needs a bit flag telling whether this advertisement was solicited (1) or gratuitous (0). Correspondingly, the flag is denoted as **S** (Solicited).

By the protocol, node B will set the **S**-bit to 1 if and *only* if this Neighbor Advertisement is being sent specifically to node A in response to a Neighbor Solicitation from node A. This is *the* information the **S**-bit conveys, making additional checks by node A redundant. For example, node A won't need to double check the IPv6 Destination Address of this Neighbor Advertisement to ensure it was its sole intended recipient.

Let's reiterate though: The arrival of an unexpected Neighbor Advertisement can never result in a new NC entry created. Regardless of its **S**-bit value, a Neighbor Advertisement can only update an existing NC entry.

Node A will have to act depending on the **S**-bit value in the NA received from B. If the **S**-bit is set (1), the link between A and B appears working fine in both directions and the NC entry can be relied upon at least for a while. It just needs to be completed with B's link-layer address from the NA and promoted to *REACHABLE* state.

For how long should an NC entry remain in *REACHABLE* state unless refreshed by signals from upper-layer protocols? To prevent nodes from locking into each other and creating concerted bursts of traffic [146], this timeout needs to be different in different nodes, e.g., set to a random value.[a] A new random value doesn't need to be generated for each NC entry though: The node can generate just one system-wide value from a certain range[b] at least every few hours and use it to time out *REACHABLE* entries until it is time a new random value were calculated [139, Section 6.3.2].

[a] Parameter *ReachableTime*, random [139, Section 6.3.2].

[b] Between $MIN\_RANDOM\_FACTOR \times BaseReachableTime$ and $MAX\_RANDOM\_FACTOR \times BaseReachableTime$, by default between 0.5×30 and 1.5×30 seconds [139, Section 6.3.2].

On the other hand, if the **S**-bit is unset (0), nothing is still known about the mutual reachability of neighbor B, but node A has at least learned B's link-layer address and can venture to send a link-layer frame with the queued packet to that address. In this case, too, node A will store the information learned in its Neighbor Cache so that it isn't lost for nothing, but the entry will progress to *STALE* state instead, to be discussed in a moment. In a nutshell, the information from a *STALE* entry has a fair chance to prove valid but it still needs to be confirmed before the entry can become *REACHABLE*.

For convenience, let's refer collectively to all states where the NC entry is filled in with a certain link-layer address as "complete" states to distinguish them from *INCOMPLETE* state and nonexistence.

A *REACHABLE* entry can be refreshed on a signal from an upper-layer protocol or the reception of additional Neighbor Advertisements with $S = 1$. Either way its timer is reset. But suppose that the entry hasn't been refreshed for a while and eventually timed out.[4] Does its timeout mean that the entry is no longer valid because neighbor B has gone missing? Under equivalent conditions, ARP would assume so and delete the entry. But in fact the timeout may mean as little as that the current conversation is over and so the neighbors just shut up and got back to some other business. For this reason node A won't let go of B's NC entry if it was *REACHABLE*: The entry will just become *STALE* instead.

[4] Parameter *ReachableTime*, random [139, Section 6.3.2].

Because a *REACHABLE* NC entry in active use is expected to be refreshed often by signals from the upper-layer protocols, its timer can be set to a much shorter value than that we know is used on ARP cache entries. So we don't need to be surprised that a *REACHABLE* NC entry, once idle, will time out and become *STALE* considerably sooner than we would expect based on our IPv4 experience alone.

A *STALE* entry contains information there is some doubt about. On the one hand, it shouldn't be used blindly, without extra checks, and so the entry remains in *STALE* state only as long as there are no more packets to pass to that neighbor. On the other hand, there is no mandatory timer on it because its information can still be useful later on.

Time passes and eventually node A has another packet to transmit to neighbor B, whose NC entry is now *STALE*. Nodes A and B are likely to be just where they were before and so node A instantly transmits the packet to the cached link-layer address from B's entry in the hope that the upper-layer protocol will confirm B's reachability soon. This event is noted by changing the entry's state to *DELAY* and starting a timer on it.[5]

What if the *DELAY* timer expires before any signal has come through from the upper layer? Even then all is not lost for the NC entry: Indeed, the packet transmitted could come from a unidirectional protocol and trigger no response from its destination, or it could be lost at some further spot in the network. To remove any doubt, node A conducts a targeted check of neighbor B's reachability by sending a *unicast* Neighbor Solicitation to B using its cached link-layer address. To reflect this event, the entry's state is changed to *PROBE*. Finally, should no Neighbor Advertisement come back from neighbor B even after several retransmits[6] and timeouts,[7] its entry will have to be deleted from the Neighbor Cache, thus restarting Neighbor Discovery for this neighbor from scratch in the hope that its new link-layer address will be revealed.

Some modern IPv4 implementations can also unicast an ARP request to refresh a complete ARP cache entry, one notable example being the *Linux* kernel [147, Section 15.3]. In fact, this technique is hardly anything new: It was approved in Item 2 of [51, Section 2.3.2.1] under the name of **unicast poll**.

[5] Parameter *DELAY_FIRST_PROBE_TIME*, default 5 seconds [139, Section 10].

[6] Parameter *MAX_UNICAST_SOLICIT*, default 3 times [139, Section 10].

[7] Parameter *RetransTimer*, default setting *RETRANS_TIMER* (1 second) [139, Section 6.3.2].

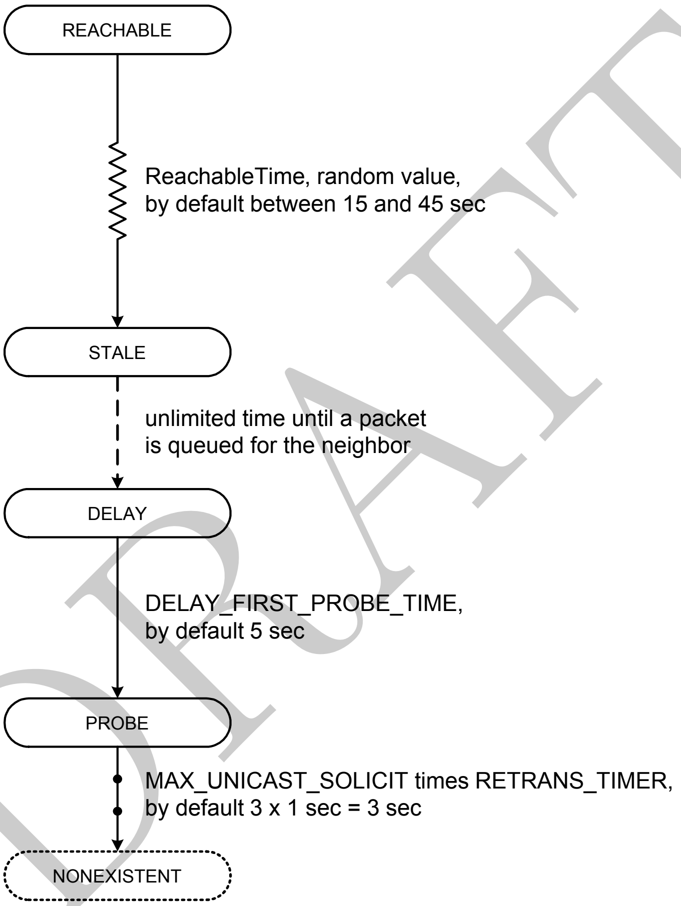

REACHABLE
ReachableTime, random value,
by default between 15 and 45 sec
STALE
unlimited time until a packet
is queued for the neighbor
DELAY
DELAY_FIRST_PROBE_TIME,
by default 5 sec
PROBE
MAX_UNICAST_SOLICIT times RETRANS_TIMER,
by default 3 x 1 sec = 3 sec
NONEXISTENT

Figure 5.4: Deorbiting an outdated Neighbor Cache entry: the timeline

Fig. 5.4 summarizes the timeline of a Neighbor Cache entry that is doomed because it has become out of date or fails to get refreshed for some other reason.

While the NC entry for neighbor B is complete but still pending in *DELAY* or *PROBE* state, more application packets can be queued for B at node A. Should node A defer their transmission until the entry is confirmed? As a matter of fact, a pause would do more harm than good here because it would disrupt the upper-layer feedback. Node A therefore should keep passing the packets to neighbor B using its cached link-layer address once B's entry is complete and thus usable. This way the upper-layer protocols, too, get a chance to confirm the NC entry: On a signal from the upper layer, any complete entry becomes *REACHABLE*. Quite obviously, such a signal can have no effect on a nonexistent or *INCOMPLETE* entry even if it comes through out of the blue simply because such an entry has no information in it yet as to the neighbor's link-layer address.

Although our original objective was just to save the resources of the link and its nodes, in the end we have managed to "build" a full-fledged mechanism to monitor the status of the neighbors the local node sends data packets to. This mechanism is known as **Neighbor Unreachability Detection** (**NUD**) [139, Section 7.3]. In essence, NUD provides a finite state machine to ND, information about neighbor reachability to be produced as the machine runs. So far, we have completed a simplified version of this machine as summed up in Fig. 5.5.

A more complete variant of the ND state diagram can be found, e.g., in [140, Page 502]. However, it will make much more sense if studied after our current work on ND has been finished.

With ARP's shortcomings eliminated from its successor, we can relax and look back on the fruitful ideas designed into the old protocol in order to carry them over to ND.

One of the things ARP got right is that a majority of ARP Requests, and now Neighbor Solicitations, open a network conversation. In other words, more often than not node A's searching neighbor B out and sending a packet to it is instantly followed by node B's having a packet to send back to node A. If A and B are just hosts, this is due to most application protocols being bidirectional; and if either is a router, the situation reflects not only the bidirectionality of a typical application protocol but also the practical symmetry of IP routing, where asymmetric routes have always been considered the exception rather than the rule owing to various complications they can

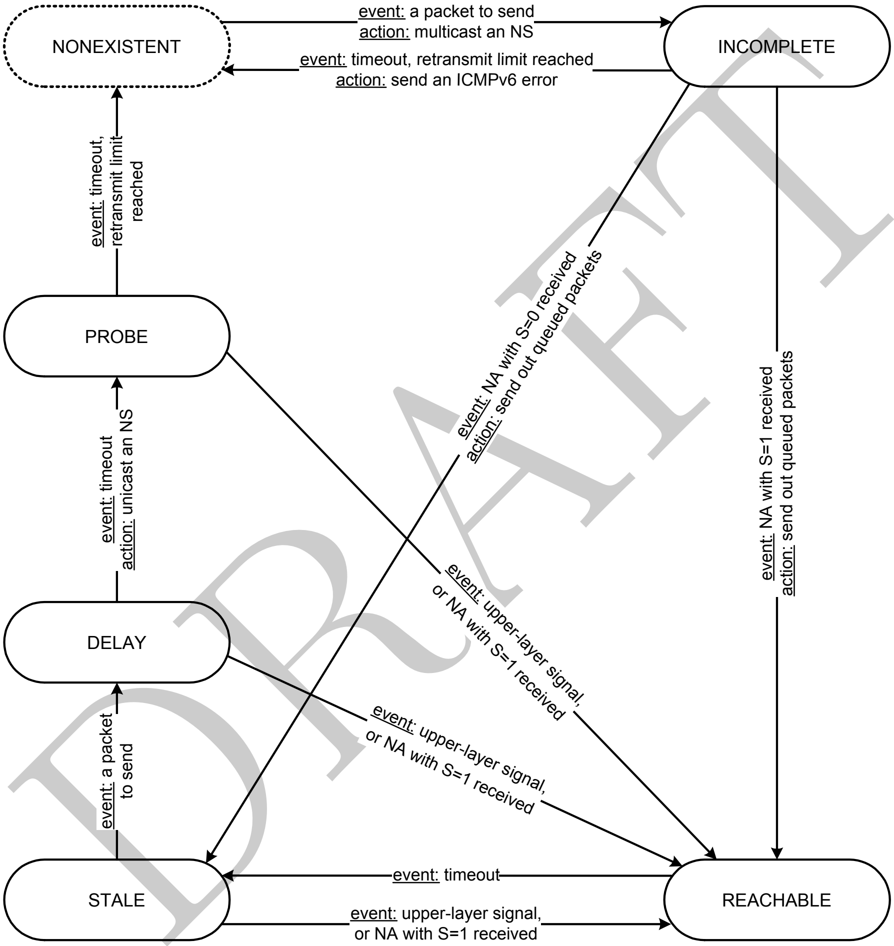

NONEXISTENT
INCOMPLETE
PROBE
DELAY
STALE
REACHABLE
event: a packet to send
action: multicast an NS
event: timeout, retransmit limit reached
action: send an ICMPv6 error
event: timeout, retransmit limit reached
event: timeout
action: unicast an NS
event: a packet to send
event: NA with S=0 received
action: send out queued packets
event: NA with S=1 received
action: send out queued packets
event: upper-layer signal, or NA with S=1 received
event: upper-layer signal, or NA with S=1 received
event: timeout
event: upper-layer signal, or NA with S=1 received

Figure 5.5: A simplified ND state diagram

bring about. To optimize for this scenario, the ARP module would try to keep a step ahead and cache the source of an incoming request if it was about this node, betting that roles would be switched soon and the request source would become the next hop of an application packet to send out.

In IPv6, what can hint at a conversation about to start is an incoming Neighbor Solicitation. If its format provides room for the link-layer address of its source (A), its target (B) will be able to create a complete NC entry for A as early as when the NS is received. By the NUD rules, that entry won't be quite reliable because only one-way reachability has been proven: $A \rightarrow B$. Such an entry deserves *STALE* state initially, but that will be enough to avoid stalling the conversation for another round of ND when node B is, in its turn, to look for neighbor A. Subsequently, providing the nodes and the link are fine, the NC entry for A will get promoted to *REACHABLE* state by the well-known procedure from Fig. 5.5 without ever delaying the application packet exchange between A and B.

Another valuable idea to borrow from ARP is that the node to respond to a neighbor discovery request isn't always the target node itself, that is, the actual owner of the IP address being looked for. The target node can be elsewhere instead of on the link and the router having a specific route to it can respond on its behalf. In this scenario, the router assumes the role of a proxy. In IPv4, it was known as ARP proxy and so its IPv6 counterpart will be referred to as **ND proxy**. Regardless of the IP version, the task of such a proxy is to attract application traffic intended for the target node and forward it off the link. The proxy therefore needs to respond with its own link-layer address to queries about the IP address proxied for, as shown in Fig. 5.6.

To adapt this scheme for IPv6, we need to go a step back and, first of all, provide for the ND proxy to receive the relevant Neighbor Solicitations since they aren't broadcast. This is as easy as having the ND proxy join the solicited-node multicast group for each IPv6 unicast address proxied for.

Should a node have to proxy for multiple IPv6 addresses, e.g., for an entire subnet, it can make use of the special mode of operation in its interface where it receives just all multicast frames, with no filter applied (*allmulti*). However, that may not work well in a switched LAN snooping on IGMP and MLD messages to learn the geography of individual multicast groups. In the latter case each Layer 3 multicast group listened to has to be explicitly joined; otherwise it can be pruned by the switches and its traffic may never reach the listener's port.

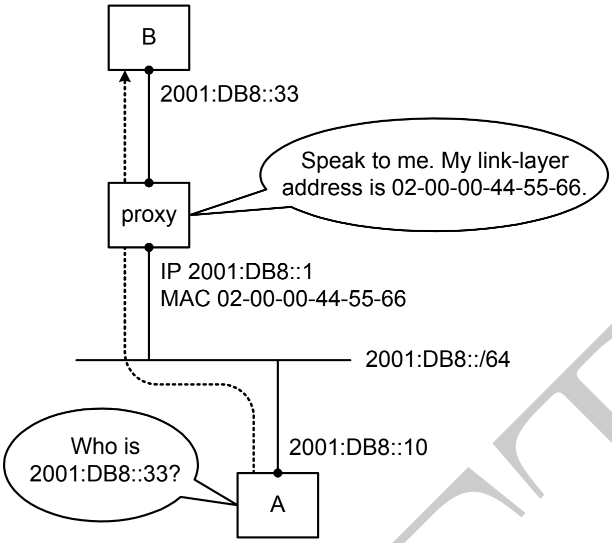


Figure 5.6: ND proxy: the basic idea

> ND proxy function must be controllable. The bare minimum will be an on-off switch or its software equivalent because ND proxying must be disabled by default.

How will this proxy scheme fit in the case where the target node is mobile and can appear on the link or off it without ever changing its IP address? Let's stick to our usual notation of B for the target node and A for our local node to look for neighbor B assuming it is on the link, as shown in Fig. 5.7.

> Although such proxy-assisted "mobility" doesn't yet provide for Internet-wide roaming, it meets the main requirement of Mobile IP: that the roaming node B stick to just one IP address regardless of which spot it is connected to the network at [148]. The alternative approach, so-called Nomadic IP, is that B's IP address is allowed to change depending on where B is connected to the network at this moment [148]. Clearly, "Nomadic IP" is much easier to do because it is based on nothing but conventional IP routing and boils down to just moving the node around and setting up a new network connection at each spot. Still, it won't provide for persistent application sessions surviving change of location—unless the application protocol itself can tolerate the endpoint's changing its IP address on the fly.

Suppose node B is initially off the link and node A has a packet queued for it. A attempts to discover B by sending an ARP request in IPv4 or

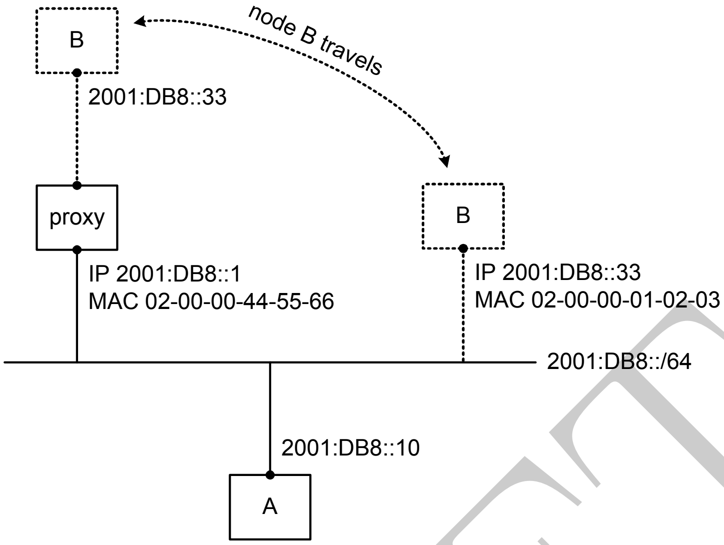


Figure 5.7: ND proxy-assisted mobility

a Neighbor Solicitation in IPv6. The proxy responds on behalf of node B with its own link-layer address. Node A has no idea it was just a proxy and transmits the packet queued for B to the link-layer address so learned. The proxy receives the packet and forwards it towards B according to its configuration. Thus node B sees no ARP request or Neighbor Solicitation from A because they aren't really neighbors at the moment.

Now let node B sneak across onto the link and become an actual neighbor of A. Once there, it can receive, and respond to, A's requests or solicitations. Meanwhile, the proxy is unaware of B's relocation and keeps responding on its behalf too. Consequently, there will be as many as two simultaneous replies (advertisements) to each discovery request by A, telling different link-layer addresses of B! What will the cache entry for B look like after A receives both? In ARP, the reply to come last would win and take over the cache entry, overwriting the link-layer address learned from the reply that had come first. The order of their delivery essentially random, in about a half of all cases the proxy's reply would win and the packets would get forwarded to where B was no more. Needless to say, this problem must be addressed in ND so as to preclude the failure mode from Fig. 5.8.

> To be precise, in IPv4 some packets, e.g., those already queued, could get sent to the link-layer address learned from the ARP reply to come first, but then the other ARP reply would come through and overwrite the cached link-layer address, thus diverting the subsequent packets.

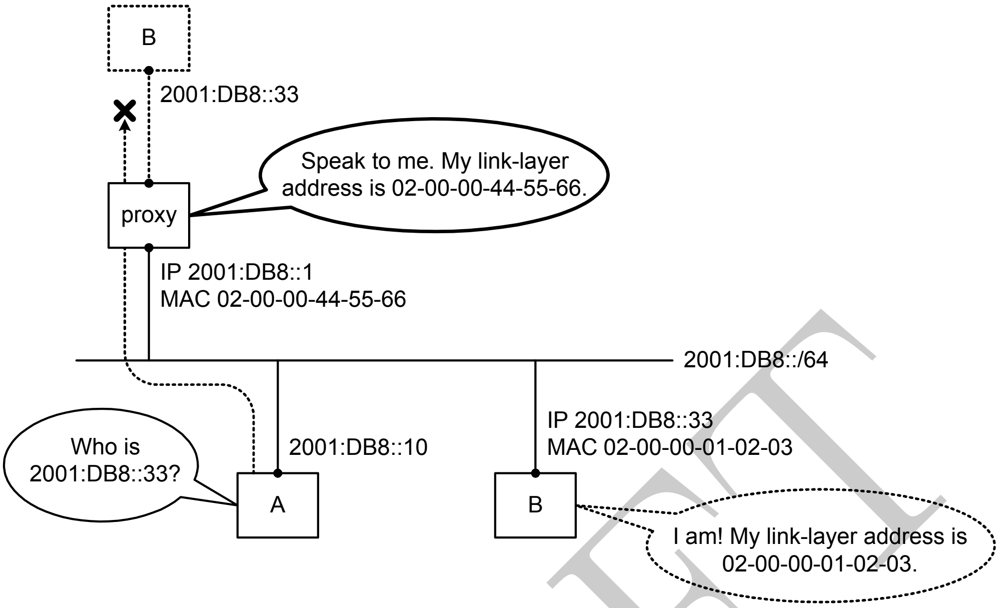


Figure 5.8: Naive ND proxy causing mobility fault

Fortunately, there are just two ranks of nodes contending for the target address. The top rank is the target node itself, the genuine owner of the unicast address *B*. There are no other nodes of the top rank since, by definition, a unicast address can be owned by no more than one node. The bottom rank is proxies, which can be multiple. Let all ND proxies have the same level of priority because they are just an auxiliary mechanism, not to be over-engineered. A single bit therefore should be enough to flag whether a Neighbor Advertisement came from the target node or just a proxy.

Let's see what strategy node *A* will need to stick to so that the target node wins regardless of the order the Neighbor Advertisements came in:

- If the target node's NA comes first:

    1. On receiving the NA from the target node, node *A* fills in the NC entry for neighbor *B* with the link-layer address learned so.
    2. The subsequent NA from a proxy doesn't change the already complete entry.

- If a proxy NA comes first:

    1. On receiving the proxy NA, node *A* fills in the NC entry for neighbor *B* with the link-layer address learned so.

2. The subsequent NA from the target node overwrites the link-layer address in the entry notwithstanding it is already complete.

In a nutshell, the target node's NA will override that from a proxy but not the other way around. The straightforward way to implement this would be to extend each NC entry with a flag bit indicating whether the link-layer address currently cached came from the target node or a proxy. But can't we go without an extra flag on each NC entry? If the NA comes from the target node itself, it will unconditionally update the NC entry, whose origin proves just irrelevant in this case. On the other hand, if the NA comes from just a proxy and the NC entry already contains a different link-layer address, it will be sufficient to mark that entry *STALE*, with its other data unchanged, and let NUD fix it up as follows.

Please bear in mind that NC entries are never created just because a Neighbor Advertisement came through: Neighbor Advertisements can only affect existing entries, complete or *INCOMPLETE*. This contrasts with how a new ARP cache entry could get born when an ARP reply was received.

Once *STALE*, the NC entry will be subject to the verification procedure from Fig. 5.5. If the cached link-layer address is current, the entry will be restored to *REACHABLE* state. This will be the case if the entry points to the target node itself, still on the link, or to a live proxy the target node should be reachable via. Otherwise the entry will expire, prompting node A to rerun ND for B from scratch. In particular, this will happen if node B has hopped off the link, to be reachable only via a proxy from that time on.

With all ND proxies having the same precedence, ND provides no means to detect which proxy the target node is actually reachable via when it is off the link and multiple proxies reply on its behalf. This problem has to be solved through appropriate configuration of the proxies involved.

Please make sure you understand that NUD can track the status of actual on-link neighbors only. Thus, should the NC entry for node B point to a proxy, its status will be that of the proxy, not node B itself. This is because NUD can provide no information on *end-to-end* reachability, its scope limited to a single link by design. This is an *architectural* point to grasp even if you are going to never use ND proxies.

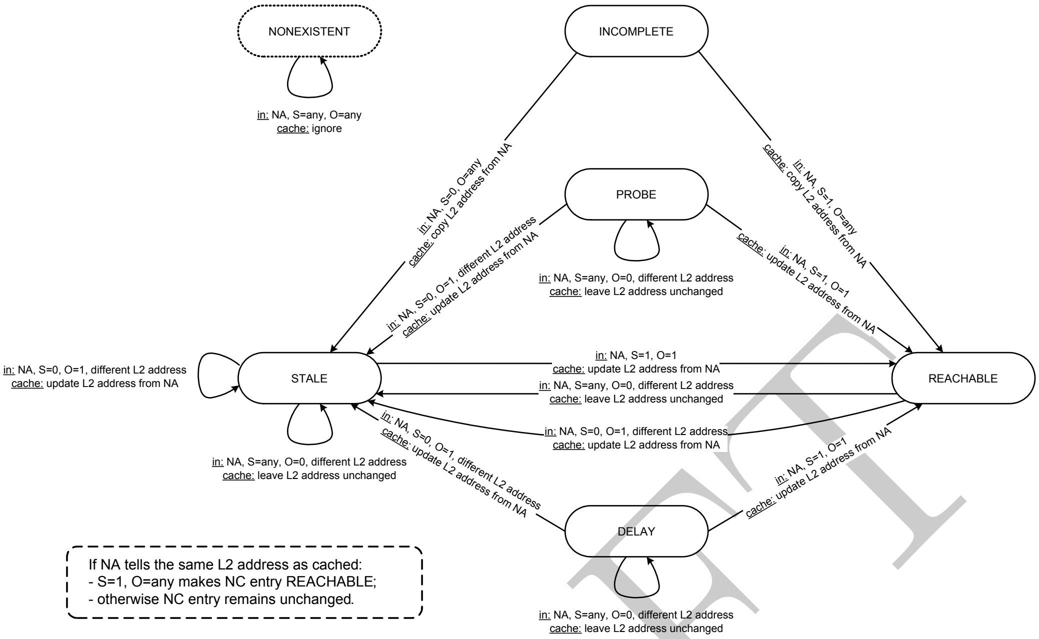

NONEXISTENT
INCOMPLETE
in: NA, S=any, O=any
cache: ignore
PROBE
in: NA, S=0, O=any
cache: copy L2 address from NA
in: NA, S=1, O=any
cache: copy L2 address from NA
in: NA, S=0, O=1, different L2 address
cache: update L2 address from NA
in: NA, S=any, O=0, different L2 address
cache: leave L2 address unchanged
in: NA, S=1, O=1
cache: update L2 address from NA
in: NA, S=0, O=1, different L2 address
cache: update L2 address from NA
STALE
in: NA, S=1, O=1
cache: update L2 address from NA
in: NA, S=any, O=0, different L2 address
cache: leave L2 address unchanged
REACHABLE
in: NA, S=0, O=1, different L2 address
cache: update L2 address from NA
in: NA, S=any, O=0, different L2 address
cache: leave L2 address unchanged
in: NA, S=0, O=1, different L2 address
cache: update L2 address from NA
in: NA, S=1, O=1
cache: update L2 address from NA
DELAY
If NA tells the same L2 address as cached:
- S=1, O=any makes NC entry REACHABLE;
- otherwise NC entry remains unchanged.
in: NA, S=any, O=0, different L2 address
cache: leave L2 address unchanged

Figure 5.9: ND state diagram providing for the **O**-bit and link-layer (L2) address change

So the new bit flag in the Neighbor Advertisement format winds up having the following meaning—a really clear one: whether to override the NC entry in case it already exists and has a different link-layer address cached in it. To reflect its purpose, the flag will be known as **Override**, or **O** for short. Let's sum up its semantics. If the **O**-bit is set in an incoming Neighbor Advertisement ($O = 1$), its information has precedence over the existing NC entry for that neighbor. First, the link-layer address from the NA is copied to the entry, replacing the one formerly cached. Second, the entry is marked as *REACHABLE* if the Neighbor Advertisement was solicited (S=1), or as *STALE* if the NA was gratuitous (S=0). On the other hand, if the **O**-bit is unset ($O = 0$), the existing NC entry isn't updated with the new link-layer address at all: Instead, a *REACHABLE* entry will become *STALE* while an entry in another complete state will remain just unchanged because it is already under verification [139, Section 7.2.5]. These rules are illustrated in Fig. 5.9.

Now, with the help of the **O**-bit and the link-layer address update rules coming with it, yet another useful idea from ARP can live on in ND. When an IPv4 node had its link-layer address changed, the node would broadcast a gratuitous ARP, in which it would either pretend to be looking for its own IP address or reply to an imaginary request from itself. Either way the existing cache entries its neighbors might have would get updated with the new link-layer address—providing the gratuitous ARP wasn't lost. We have already refreshed our memory on the background of this trick: By the ARP rules, the first thing a recipient of any ARP message should do was look the source up in its own ARP cache and, if found, update the link-layer address cached [144]. Although this trick worked reasonably well in IPv4, now we aim to have each ND message type do its specific job. Fortunately, an IPv6 node already has right to send a Neighbor Advertisement any time it feels like doing so, and its only duty in this respect is to clear the **S**-bit in such a gratuitous NA ($S = 0$).

> A gratuitous ARP request was also used to detect IP address conflicts, but that function will be discussed in Section 5.4.1.

An IPv6 node therefore can send a gratuitous NA about its link-layer address change. The flag settings appropriate for this NA will be $S = 0$ and $O = 1$ because the NA is unsolicited but it comes from the top authority, that is, from the node itself. And what IPv6 Destination Address does it need to be sent to? The node having no idea which neighbors have cached it, its gratuitous NA will have to be multicast to all IPv6 nodes of the link, that is, to the group *FF02::1*. For a better chance of its delivery to all nodes concerned, the message can be repeated a few times[8] with short pauses in between.[9]

> Some operating systems also send a gratuitous Neighbor Advertisement just after an IPv6 address was assigned to the interface. One example spotted was *Cisco IOS* 12.4.

Now that the first batch of problems posed by IPv6 Neighbor Discovery has been solved, the requirements to the ND message formats are finally known and it is time those formats were worked out.

By our plan, the focus of the Neighbor Solicitation message is the IPv6 address being looked for. From the experience just gained, it isn't hard to see that such an address will need to appear in different ND message types simply because neighbors are identified by their IPv6 addresses. So let's refer to the IPv6 address a particular ND message, regardless of its type, is about as the **target address**. Consequently, the core element of the Neighbor

[8] Parameter *MAX_NEIGHBOR_ADVERTISEMENT*, default 3 times [139, Section 10].

[9] Parameter *RetransTimer*, default *RETRANS_TIMER* (1 second) [139, Section 6.3.2].

Solicitation will be the **Target Address** field as shown in Fig. 5.10 [139, Section 4.3].

The Neighbor Solicitation has required no flags. At the same time, we planned that it could include the source link-layer address to optimize ND for bidirectional communication, with the source IPv6 address available from the IPv6 header of the NS packet. Can the source link-layer address be good for anything else to justify its inclusion in the Neighbor Solicitation format?

In fact, we paid so much attention to the rules and procedures for the soliciting node that completely forgot about those for the solicited node. Not that they are complex, but one detail of importance was overlooked: To respond to a Neighbor Solicitation with a unicast Neighbor Advertisement, the soliciting node's link-layer address needs to be known. Quite obviously, ND can't be used to discover it because the nodes already are in the middle of ND and a deadlock would ensue. The alternative will be to have the soliciting node include its link-layer address in the Neighbor Solicitation. Of course, the link-layer frame around the Neighbor Solicitation should contain its source address, but historically access to the frame header from the IP layer can be problematic due to implementation quirks. This consideration warrants providing a copy of the source link-layer address in the Neighbor Solicitation body.

So it turns out that a typical Neighbor Solicitation will simply *need* to have its source link-layer address in it. There is going to be just one major exception from this rule: In Section 5.4.1, a special Neighbor Solicitation will be required, having the unspecified IPv6 Source Address and indicating no source link-layer address. It is easy to guess that such an odd Neighbor Solicitation will be needed to work around some kind of chicken-and-egg problem.

Also, there is a case where including the source link-layer address in a Neighbor Solicitation can be recommended but not necessarily required: when the Neighbor Solicitation is unicast [139, Section 4.3]. Indeed, such a message can be sent to a neighbor only when its link-layer address is already in the Neighbor Cache and so there is a good chance that the neighbor also has the soliciting node's link-layer address cached [139, Section 7.2.2]. The neighbor is supposed to have learned it from the very first Neighbor Solicitation, which had to be multicast and include its source link-layer address.

Verify that this optimization will work fine even if the application protocol running from the local node to the neighbor is unidirectional, such as Syslog or Netflow.

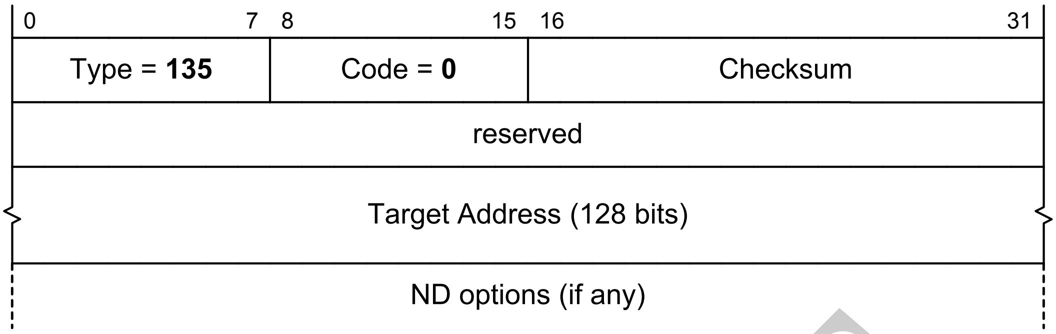


Figure 5.10: Neighbor Solicitation format

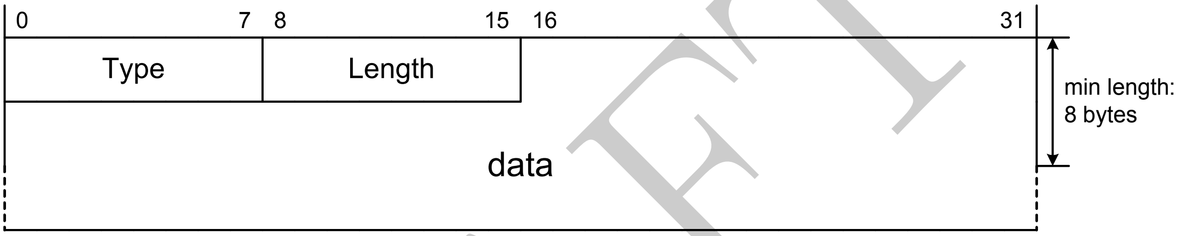


Figure 5.11: ND option format

Should the neighbor happen to have no NC entry for the soliciting node, e.g., because it has just rebooted, it will have to use Neighbor Discovery and send out a multicast NS [139, Section 7.2.4]. Although recursive discovery is generally dubious, in this case the protocol won't loop forever because the multicast NS must carry its source link-layer address in it.

Because the source link-layer address will be optional after all, albeit carried by most Neighbor Solicitations, ND options need to be introduced now.

Let ND options be based on TLV encoding as shown in Fig. 5.11, namely: the **Type** byte followed by the **Length** byte followed by variable-length data whose format is controlled by the option type [139, Section 4.6]. For a change, the ND option length value will include the **Type** and **Length** bytes and so never be zero in a valid option; its unit will be the 8-byte word. The presence of options in an ND message can be detected by checking where its end is: Should there be extra bytes after its fixed part, those are ND options. As for how to determine ND message boundaries, they can be found by the guideline from Section 4.3 common to all ICMPv6 messages because ND is just a subset of ICMPv6—remember?

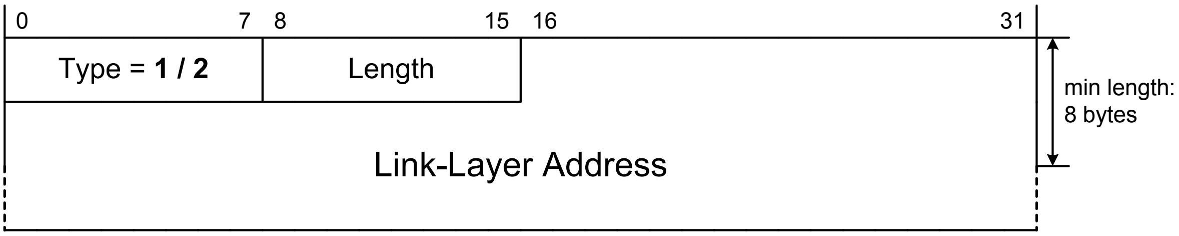


Figure 5.12: SLLA (*Type* = 1) and TLLA (*Type* = 2) options: generic format

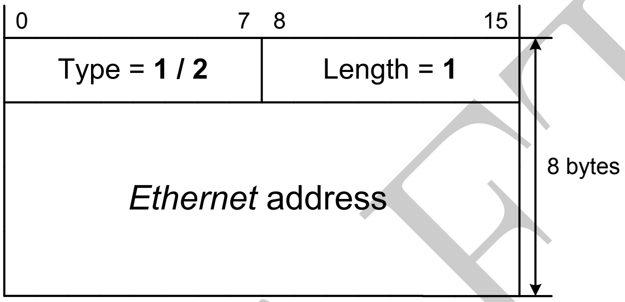


Figure 5.13: SLLA (*Type* = 1) and TLLA (*Type* = 2) options for *Ethernet*

The **Source Link-Layer Address Option** (**SLLA** or **SLLAO** for short) to appear in Neighbor Solicitations is supposed to contain just a link-layer address and will have a Type of 1, resulting in the trivial format shown in Fig. 5.12 [139, Section 4.6.1]. Then, for example, the MAC address *02-03-04-05-06-07* will be encapsulated in an SLLA thus: {0x01, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07}.

> Each link-layer technology suitable for IPv6 is supposed to have an RFC document specifying the details of how IPv6 will operate over links of this type. One such detail is the exact format of the ND link-layer address options. E.g., see [121, Section 6] for *Ethernet*—illustrated by Fig. 5.13.

> Unlike in ARP, the type of a link-layer address is never indicated explicitly in ND, the assumption being that it can be figured out unambiguously from the type of the link the ND message came from.

The Neighbor Advertisement message is to carry the actual information about a neighbor, or about a node behind a proxy, both known to us under the umbrella term of the target. This information is to provide for the mapping from the target's IPv6 address to its apparent link-layer address. Thus, in its first draft, the NA format needs to accommodate those target addresses. Although the NA source address is told by the IPv6 header, it may not be the same as the target address, e.g., if the NA was sent by a proxy rather than the target. For the proxy to never have to use a forged source address, the target address will be stored separately, in the NA message body [139, Section 4.4].

The formats of NA and NS can be made quite similar so as to reduce implementation complexity. For that, the target link-layer address will be stored in the NA message as an option instead of a mandatory field. When a multicast NS is being responded to, this option must be included in the NA so that the target link-layer address is made known. On the other hand, if the triggering NS was unicast, it can be safe to assume that the soliciting node already has the current link-layer address of the target cached; otherwise the unicast NS wouldn't have been received in the first place. In this case the target link-layer address can be omitted from the NA albeit the benefit from doing so will be small, if any at all. This ND option, known as **Target Link-Layer Address Option** (**TLLA** or **TLLAO**), differs from the SLLA option only in its type value, which will be 2 for TLLA—see Fig. 5.12 and Fig. 5.13.

> A Neighbor Advertisement with no TLLA can also be encountered in non-standard applications of ND. For example, *Cisco IOS* 12.4 sends an NA with $S = 0$, $O = 0$, and no link-layer address after bringing IPv6 up on a PPP interface—probably because the PPP interface has no native link-layer address on it. The destination address of that NA is all IPv6 nodes of the link, *FF02::1*. Apparently, with that NA the *IOS* stack just wants to make its local IPv6 address known to the far-end peer.

Besides, several flags were planned in the NA format. In addition to the already familiar **S** and **O** flags, let's allocate a bit for the **R** flag, to be introduced in Section 5.4.2. The resulting format is shown in Fig. 5.14.

So our conceptual IPv6 node is now able to form a Neighbor Solicitation or Neighbor Advertisement message when necessary. And which interface is that message to be sent through if the node happens to have more than one interface?

A Neighbor Solicitation will go out through the interface into the link where neighbor discovery is to take place. This was an important point we made at the beginning of this section: that neighbor relation is defined

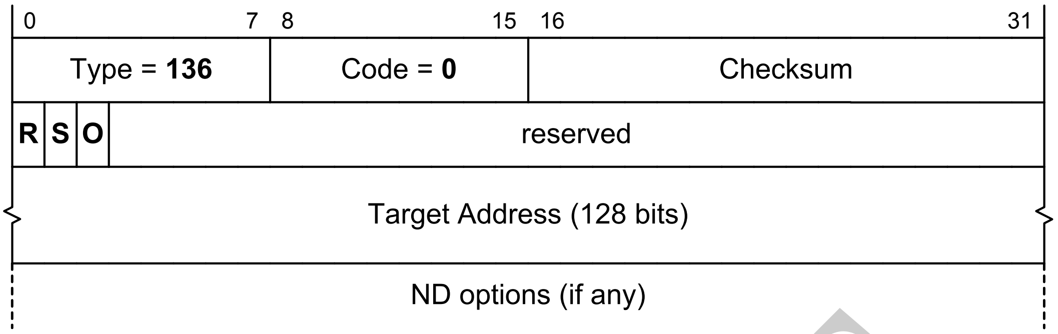


Figure 5.14: Neighbor Advertisement format

between nodes only on a particular link. Consequently, neighbor discovery isn't conducted globally but only after the soliciting node worked out which link it would be interested in. Neighbor discovery, in its turn, is triggered by a queued packet, whose next hop and egress interface are supposed to be known from a routing decision made by the stack at an earlier stage. It will be quite natural therefore to send the NS out via the same interface the data packet is queued on. This way the relevant link will be chosen automatically just because only one link can be connected to an IP-compliant interface at one time.

As for a Neighbor Advertisement, the egress interface choice depends on whether this message is solicited or not. If it is being sent in response to an NS ($S = 1$), it can be transmitted from just the same interface the NS was received on as means to ensure it ends up in the correct link context. On the other hand, if the NA is unsolicited ($S = 0$), the link context should be known from other criteria. For example, if this is an advertisement about link-layer address change, it will need to go out the interface the address has changed on.

Great, now neighbor discovery can actually start and proceed to completion using the message formats and handling rules available. What information apart from the target link-layer address can be learned by node A when it successfully discovers neighbor B? One such fact to infer is that the link was passable in both directions, i.e., from A to B and back, at least at the time of the neighbor discovery session. Thus ND can confirm mutual reachability of the neighbors, and NUD will help to keep this information current. Should just one direction get blocked, resulting in a "half-link", the neighbor will be reported as unreachable. That is to say, ND supports only symmetric links for now [139, Section 3.2].

Someday ND may be extended to accommodate asymmetric links as well [139, Section 3.2]. Of course, that will hardly happen until there is a sufficient demand for such links in IPv6 environment.

This concludes the first part of our work on IPv6 neighbor discovery, which was focused on unicast address resolution. In later sections we will encounter a few more problems whose solutions can be more elegant if put under the same umbrella of the IPv6 **Neighbor Discovery Protocol** (**NDP**) [139, Section 3]; but let's cross those bridges when we come to them.

Our final remark here will be about the improvements pioneered by IPv6 now gradually making their way back into IPv4. As a case in point, the modern versions of *Linux* [147, Section 15.3] and *MS Windows* [149] have had their ARP modules extended with an equivalent of the ND state machine including upper-layer feedback, fine cache control based on NUD, and other features not in the traditional ARP implementations.

5.2 The Host, the Link, and the Subnet in IPv6

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
>
> J.W. Riley

> CROWD: A witch! A witch! A witch! We have got a witch! A witch!
> VILLAGER #1: We have found a witch, might we burn her?
> CROWD: Burn her! Burn!
> BEDEMIR: How do you know she is a witch?
> VILLAGER #2: She looks like one.
> BEDEMIR: Bring her forward.
> WITCH: I'm not a witch. I'm not a witch.
> BEDEMIR: But you are dressed as one.
> WITCH: They dressed me up like this.
>
> Monty Python and the Holy Grail

Until now, we have mostly spoken of IPv6 nodes in general, only slightly differentiating between the host and the router, and that made sense as long as the most basic mechanisms of the IPv6 node were discussed.

It won't hurt to recap the definitions of the IP host and router here. Ideally, the host only originates and consumes IP packets while the router is concerned exclusively with forwarding hosts' packets. Of course, in reality a router has to include as much of host aspect as required to handle control messages, support remote management, and participate in routing protocols.

Still, we are bound to face the following question: In the IPv6 reality, can we hold on to our IPv4 experience and keep regarding the differences between the host and the router as insignificant in every aspect but transit packet forwarding?

It can't be denied that the two node types have de facto converged in IPv4, with a typical implementation providing router mode as well as host mode off the same code package in which the current mode is selectable at run time with a software toggle switch to enable or disable the forwarding of transit traffic. This feature is backed by the algorithms and data structures of the IPv4 stack being the same regardless of the node type. In particular, either IPv4 node type:

- employs a routing table to map remote prefixes to next hops—compare [51, Section 3.3.1.2] and [84, Section 5.2.4.3];

  > A full-fledged routing table was declared optional for an IPv4 host [51, Section 3.3.1.2], but in practice it enjoys ubiquity.

- works out if a remote node is a neighbor by matching the subnet prefixes on its local interfaces with the remote address—compare [51, Section 3.3.1.1] and [84, Section 5.2.4.2].

  > We disregarded unnumbered point-to-point interfaces since they require no neighbor discovery.

At the deepest level, this host-router convergence is based on a certain model of how IPv4 hosts and routers are expected to interact within one subnet. According to that model, it is up to the host to make a decision which neighbor to transmit an outgoing packet to, while the router will just forward any transit traffic the host has handed off to it. For this scheme to work, the host needs to know the logical topology of the link that the subnet in question is assigned to. Only then will the host be able to ensure that the next hop chosen for a packet is a real neighbor. This point is of paramount importance for successful transmission because, should the host get it wrong and mistake for a neighbor a node reachable only via a router, a direct transmission of the packet to it will be doomed to fail.

> Note how we are relying on another fundamental model here: that an IP subnet never maps to more than one link [21, Section 2.1].

Let's reexamine the IPv4 neighborship test from this perspective. Suppose node A is about to send a packet with a Destination Address of B over its interface I having these addresses assigned: *192.0.2.3/28* and *203.0.113.131/28*. Thus A's viewpoint is that the link connected to interface I has two IPv4 subnets on it: *192.0.2.0/28* and *203.0.113.128/28*. If the destination address B is equal to, say, *192.0.2.8* or *203.0.113.136*, node A's immediate conclusion will be that it is a neighbor's because it has a directly connected subnet prefix in it. That being so, A's IPv4 stack can just invoke the underlying link layer to get the packet over to its destination B. On the other hand, none of the addresses *192.0.2.150*, *198.51.100.1*, and *203.0.113.2* appears to be a neighbor's and, consequently, such destinations are reachable from A only indirectly, via a router.

In reality the link in question can have additional IPv4 subnets assigned to it, unbeknownst to node A because A has got no address from them.

This IPv4 logic is based on the assumption that all nodes of the link are neighbors needing no router to send packets to each other. It is often taken for granted because broadcast links such as the ubiquitous *Ethernet* meet this requirement and so do point-to-point links. But is it really universal or can there be links that differ?

The networking practice proves that some cases call for such unconventional links, generally known as **Non-Broadcast Multi-Access**, or **NBMA** for short. This term may not be quite accurate because NBMA links often have limits on unicast communication as well as on broadcasting and multicasting. A prime example of an NBMA link is that of star topology, aka hub-and-spoke, as shown in Fig. 5.15. Its main feature is that its leaf nodes can speak only to its central node, aka the concentrator, and never to each other directly. Thus the concentrator is a neighbor to each leaf node and vice versa but no leaf nodes are neighbors to each other.

Two classic NBMA link technologies were Frame Relay and ATM, where each spoke was implemented as a virtual circuit.

Modern NMBA star examples are broadband cable, cellular, and xDSL networks. Unlike *Ethernet* or *WiFi*'s, their concentrators do no link-layer switching. For that reason, their leaf nodes can communicate only via an IP router, which is usually built into the concentrator unit.

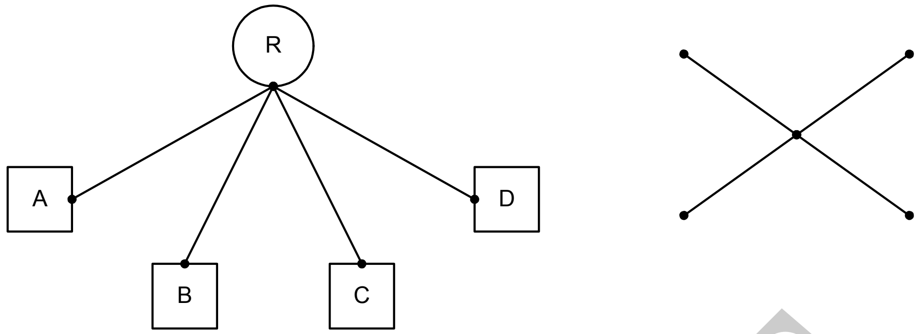


Figure 5.15: NBMA star and its topology graph

Even an *Ethernet* LAN can be reduced to star topology using Private VLAN. That can be done, e.g., for the sake of greater security, in order to control communication between leaf nodes in a centralized manner without having to give up on the simple scheme of just one LAN and one IP subnet.

If we were to conduct a purely theoretical analysis of NBMA links, we would perhaps have to consider two extreme cases. One is free multi-access that lacks only broadcast/multicast capability, so each node still can unicast to any other node provided the link-layer destination address is known. At the other end of the spectrum is the point-to-multipoint star, where even unicast communication is severely constrained.

However, many practical cases see a mix of those where only select pairs of nodes are neighbors. Because of that, we are now interested in the more general scenario where the capability of nodes to unicast or multicast over the link is restricted in a certain way. Our idea of NBMA is such that this model, too, can be considered under the NBMA umbrella.

The reader will be right to point out here that such a star is just a superposition of multiple point-to-point links rather than one proper link—in theory. However, from a practical standpoint, to treat each spoke as a separate link would be a suboptimal choice. Suppose a concentrator in a rapidly expanding broadband network has a few thousand subscribers already connected to it, their number increasing by hundreds each day. If the straightforward approach were employed and each subscriber connection were to be treated as a point-to-point link, today the concentrator would require thousands of interfaces and some day soon their number could reach the million mark! A theoretician will see no problem with that, but in reality the number of interfaces an IP node has to handle can't get arbitrarily large.

The number of interfaces in an IP node should be capped not just because its network stack implementors were lazy and used a poorly scaling algorithm. The more fundamental reason is that the interface is a central control and management object of the stack, having a configuration or state structure in nearly each stack component. The latter argument can still be countered by pointing out that RAM size is effectively unlimited in modern computer systems while any number of similar interfaces can be configured in a scalable manner using a single configuration template. Yet another strong argument in favor of limiting the number of interfaces is that it is convenient to specify a network security policy with respect to the ingress and egress interfaces of a packet, with the total number of rules in such a policy proportional to the number of interfaces squared. But even it can be parried with a reference to the security zone concept where many interfaces are grouped into just a few zones to define policies between. However, in the end it proves easier just to put a cap on the number of interfaces than to keep overcoming, or working around, one problem after another.

Hence a demand for the artificial solution where a single IP subnet is assigned to what a prudent reader might call multiple disjunct connections, thus transmuting them into one IP link by the fundamental rule of TCP/IP that each subnet maps onto just one link [21, Section 2.1]. Although something about it might smell fishy at first, let's take time to see what mechanisms will be needed to implement this trick in IPv6. We have got the luxury to be free to experiment, and the result, if successful, will be of undoubted practical value. As usual, we won't go without encountering and solving more exciting and fundamental problems in the course of our "research".

The new game rules are such that a link can have a non-trivial topology where all nodes are no longer neighbors. Formerly, when the link was assumed to be fully connected, its trivial topology was built into the IPv4 neighborship test. As of now, link topology information will have to be specified explicitly, e.g., as a neighborship matrix in which 1 marks a pair of neighbors and 0, of non-neighbors. Such matrices for the NMBA star from Fig. 5.15 and a broadcast link with the same set of nodes are shown in Table 5.1 and Table 5.2, respectively. The question is, will the link neighborship matrix or its equivalent have to be copied beforehand into each node's configuration for IPv6 to run over that link?

Neighborship matrices of relevance to TCP/IP are symmetric because neighbor relation is commutative by the model we adopted in Section 5.1.

Table 5.1: Star neighborship matrix—see Fig. 5.15 for topology

	A	*B*	*C*	*D*	*R*
A	1	0	0	0	1
B	0	1	0	0	1
C	0	0	1	0	1
D	0	0	0	1	1
R	1	1	1	1	1

Table 5.2: Broadcast link neighborship matrix

	A	*B*	*C*	*D*	*R*
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
R	1	1	1	1	1

Let's start off with an IPv6 link having just one router on it, the other nodes being hosts. If its topology were to be truly arbitrary, the topology information would probably have to be distributed across all nodes and a suitable protocol would be required unless done by hand.

However, not all topologies will make sense for TCP/IP: Usable are only those in which the router is a neighbor of each host. Otherwise, for the link to remain one, some hosts would have to forward other hosts' traffic, which they can't by definition. If put in the basic graph theory language, a star graph with the router at its center will be a subgraph of any link topology relevant to TCP/IP, as illustrated by Fig. 5.16.

As long as this is the case, the IPv6 router will be the natural source of link topology information because it can handle a virtually unlimited number of hosts, up to 2^{63}, and it is known to be able to deliver such information directly to each host using only link-layer means. Now we have just to develop a suitable mechanism for the router to convey the information to the hosts.

What settings will be absolutely necessary for an IPv6 host with a single interface to start accessing the network under the given conditions? It isn't hard to see that the bare minimum is the host's own address and the router address. At the same time, the subnet prefix length is no longer as important as in IPv4 because it has ceased to carry link topology information.

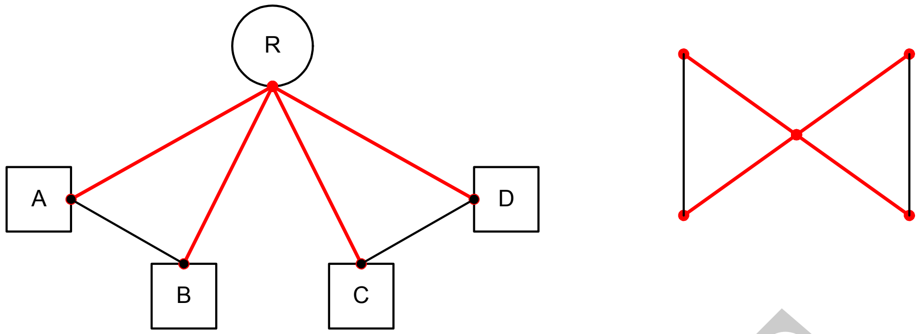


Figure 5.16: NBMA "butterfly" with a star subgraph in it

Now we are speaking about a host address of a scope larger than link-local. The host will also need at least one link-local address for its interface, as was said in Section 2.5.

At the same time, the router address can be link-local, which was also discussed in Section 2.5.

We have certainly kept in mind since Section 2.6 that an IPv6 address is always assigned to a specific interface, and in the current scenario the interface into the link in question is implied whenever we say, "host address" or, "router address".

With only this basic information available, the host can do nothing but send *all* of its outbound traffic via the router. Consequently, to the host it will be the **default router**.

An obvious exception is the loopback traffic sent by the host to its own addresses: It isn't supposed to leave the host boundaries in the first place.

What with our IPv4 experience, the following interpretation can be tempting: that the IPv6 host prototype has got only a default route, *::/0*, and no connected subnet route. However, the point is to remove the routing table altogether from the essential data structures of the IPv6 reference host. One cause for that is to simplify the minimal implementation, to be found in embedded or otherwise constrained systems. The IPv6 host is effectively being relieved of all functions known to be available from the router. An implementation can opt to include them, but they aren't mandatory and their omission won't affect interoperability.

To get the traffic going through the router, its link-layer address will first need to be found out and, depending on the link type, this may have

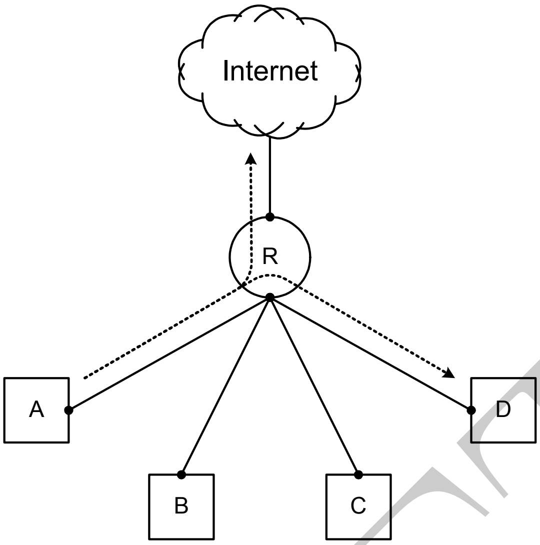


Figure 5.17: Using the default router to access both the NBMA link and the Internet

to involve Neighbor Discovery. However, the ND step won't be a problem for the host even in the face of a complex link topology because the router is guaranteed to be a real neighbor of the host as long as the link is IP-compliant. Thus a detailed knowledge of the link topology won't be required for the host to start sending packets via the router.

Having all link traffic through the router may not seem the best solution in general but it will be a perfect fit for the popular case of star topology, as illustrated in Fig. 5.17. Indeed, the hosts on an NBMA star already have no way to exchange packets but via the router, although they are nominally on the same link.

Now let's add some real complexity to the link topology in the form of direct connections between some pairs of hosts. For example, consider the "butterfly" topology shown in Fig. 5.16, where two independent host pairs have been connected. The neighborship matrix for this topology will be as shown in Table 5.3. A star centered at the router will necessarily be a subgraph of the "butterfly" to keep it IP-compliant, and so the basic operation mode where all traffic from the hosts traverses the router will remain as possible. On the other hand, it will no longer be optimal because hosts A and B are now given a chance to speak to each other at link layer, bypassing the router and reducing its load, and so are hosts C and D.

By our assumption, router R has been informed of the link topology change. We can imagine that the updated neighborship matrix has been

Table 5.3: "Butterfly" neighborship matrix—see Fig. 5.16 for topology

	A	B	C	D	R
A	1	1	0	0	1
B	1	1	0	0	1
C	0	0	1	1	1
D	0	0	1	1	1
R	1	1	1	1	1

uploaded to it in some form. So router R is aware that A and B as well as C and D are now pairwise neighbors. How can the router pass this information on to, say, host A so that the latter can start sending packets for B directly to their addressee? And when should the router do so? Will a new protocol have to be created for A to solicit the detailed topology information from R?

In fact, this seemingly complex cluster of problems can be solved with a mere **Redirect** message naturally implemented under the ND umbrella [139, Section 4.5]. The scenario can be as follows:

1. Host A passes a packet with a Destination Address of B to its default router R and creates a Destination Cache entry mapping destination B to next hop R: $B \to R$.

2. Router R detects there is a direct connection from A to B in the link neighborship matrix.

3. Router R still forwards the packet to host B as expected.

4. Afterwards, to optimize the subsequent traffic, R sends a Redirect message back to host A indicating that host B should be reachable directly.

5. Host A complies with the Redirect by changing the next hop cached in B's DC entry to B, so the resulting entry will look like this: $B \to B$.

6. As long as the "$B \to B$" entry is in its DC, host A sends packets for destination B directly to host B.

7. When the DC entry expires, the procedure is just repeated.

8. Should the Redirect message be lost and fail to reach host A, the procedure will be retried as soon as host A has another packet for destination B.

9. Finally, if the Redirect message accidentally gets duplicated by the network, the copy to come second will effectively do nothing because Redirect is an idempotent operation.

It goes without saying that router R shouldn't issue a Redirect on *each* packet received from A if the latter seems to ignore the Redirect messages and keeps sending packets for B through R. The robustness and security considerations demand that Redirect messages be rate-limited [139, Section 8.2]; but this requirement isn't news to us: It applies to nearly all ICMPv6 message types.

This solution has several clear advantages. First, it builds on a data structure the IPv6 host already has: the Destination Cache. Second, all it requires is a single new ND message type with rather simple semantics. Third, it can tolerate packet loss and duplication. Fourth, even with the neighborship matrix as known to router R changing on the fly, host A will be able to keep its idea of the link topology updated thanks to the operation of its local Destination Cache. For instance, should the A to B connection vanish, host A will revert to sending packets for B via router R as soon as the present DC entry expires.

Topology updates can occur even sooner if the Destination Cache isn't just expired by the wall clock but also ties in with the Neighbor Cache and the NUD mechanism from Section 5.1. E.g., when the A to B connection goes, NUD will soon indicate that neighbor B is no longer reachable—see Fig. 5.4 in page 191. Then it will be the right moment to review the Destination Cache as well, removing the entry with a cached next hop of B. We will extend this idea later in this section.

The Redirect-based mechanism is so simple and universal that it deserves to be part of any IPv6 host. Besides, it isn't too taxing on the host resources, which is a decided plus given the trend to use TCP/IP in industrial control and management networks, where individual hosts are underpowered both CPU-wise and watt-wise.

By all appearances, the packet sending rules for the IPv6 host are going to be significantly different from those de facto adopted by the modern IPv4 host. We should reflect this difference in the definitions and standard procedures so as to avoid confusion and interoperability issues in the IPv6 implementations to follow.

To start with, the basic problem statement for the host remains the same regardless of the IP version:

Given: The Destination Address of an outbound IP unicast packet.

Problem: Determine if the interface having the given address is reachable directly at link layer.

What does link-layer reachability of the remote interface I mean to the local host H? Evidently, it means that host H can exchange suitable link-layer frames with interface I using one of its own local interfaces, J. Through that, H can speak at link layer with the node owning interface I. Indeed, each network interface, by definition, a) is connected to just one link, and b) belongs to just one node. (For our purposes, a disconnected or otherwise unconfigured interface isn't ready and so doesn't exist for us.) A required but insufficient condition for link-layer reachability is that host H have some interface J connected to the same link L as interface I. The complemental condition is that interfaces I and J be actually able to transmit data to each other over link L. If the latter can be taken for granted in conventional LANs such as *Ethernet*, in NBMA links all depends on the particular topology built.

Interestingly, the notion of loopback traffic allows to extend this construct to accommodate the case where the destination interface I is owned by the local host H. The trivial solution for this case is as follows: $J = I$. However, if host H has multiple interfaces connected to link L, there can be other, non-trivial, solutions where $J \neq I$.

When the answer to the basic question at issue is affirmative, the host will resolve the destination address using ARP or ND, depending on the IP version, and transmit the packet to its destination directly, interface to interface. Otherwise the host will need to rely on a suitable router.

Now, what is completely different across the IP versions is how the host is to work out the answer to the basic problem of packet sending.

As we have recalled at the beginning of this section, the IPv4 host would assume that each address from its own subnet should be reachable directly, if at all. Such an address was said to be directly connected and had just two states possible: either unassigned, or present on an interface reachable at link layer from the local host. The other IPv4 addresses weren't directly connected, being reachable only via a router instead. The IPv4 host would classify destination addresses using a trivial bitwise arithmetic involving its own subnet prefixes. As a byproduct in a multihomed host setup, if the test for direct reachability came out positive for more than one local interface, the most specific subnet prefix matched so would be on the interface to send the packet through.

By contrast, the IPv6 host won't trust the apparent subnet information because it doesn't always reflect the underlying link topology. Unable to work out a priori if an arbitrary destination address is owned by a neighbor, the IPv6 host has to rely on some additional input to make an educated guess for each destination individually. Quite naturally, until such information is obtained, *the IPv6 host must assume by default that the destination interface is unreachable at link layer and the destination address is* ***off-link*** [151, Section 3]. For example, if the IPv6 host interface has the address *2001:DB8:A:B::42/64*, the host will have to rely on its default router when speaking to *2001:DB8:A:B::1234* as well as to *2001:DB8:C:D::5678*, notwithstanding that the former destination is on the same subnet.

> Please take special note of this rule, for it can be quite surprising to those coming from IPv4 background.

What are the criteria that can provide the IPv6 host with non-default information regarding destination reachability? First, its own addresses are directly reachable because the host shouldn't need a router to be able to speak to itself. Second, if the default router address is configured on the host, it is also supposed to be directly reachable by what was said in this section. In turn, that configuration setting enables the host to send packets to all destinations, including the off-link ones, by using the default router as the catch-all next hop. While doing so, the host can receive a Redirect from its default router indicating that destination B is reachable through next hop B: $B \to B$. This will mean that address B is also directly reachable from the local host. All these criteria can indicate to the IPv6 host that the destination in question is **on-link**.

The model in which all destinations are off-link by default is quite new and there has been no extensive experience with it yet, so it is unsurprising that even some RFC standards had to be amended to comply with it. For example, the current standard on ND required originally that an IPv6 address be considered on-link if it was the Target Address of an incoming Neighbor Advertisement or the IPv6 Source Address of any ND message received [139, Section 2.1]. However, such a behavior would be contrary to the basic premise that the only authoritative source of link topology information is the router. Moreover, it could enable an on-link attacker to intercept traffic intended for an off-link host. For these reasons, this shortcut has been prohibited by a more recent amendment [151, Item 3, Section 3].

At the same time, it isn't forbidden for an IPv6 host to populate its Neighbor Cache in advance based on the incoming ND messages as long as on-link status of the target isn't inferred from that. For instance, if a Neighbor Solicitation with an SLLA option has come through, the NC entry for its source will be created in *STALE* state by the local host [151, Section 3][139, Section 7.2.3] in order to avoid pausing for a full ND round later, as was discussed in Section 5.1. Still, using that entry for direct transmission won't be allowed until the router indicates that the target is actually on-link and a corresponding DC entry is created.

Only now, having understood and articulated the principal difference between the neighborship test algorithms as adopted by IPv4 and IPv6, can we proceed to optimize the IPv6 one with no risk of missing any fundamental points.

In practice *some* IPv6 address ranges can be known to be on-link with respect to a given host. Those may include:

(1) all of the link-local addresses, *FE80::/10*;

(2) subnets assigned to genuine broadcast links;

(3) ranges smaller than a subnet in which the link topology ensures direct reachability from the given host.

Besides, multicast addresses are always assumed to be on-link, but that is so because IP multicast is propagated through the network using a technique entirely different from that utilized by IP unicast. Namely, multicast routers just listen to the groups to be forwarded off link and so, from a multicast source's point of view, they are no different from mere group members residing on the same link. More will be said on this topic in Section 6.4. In addition to that, IPv6 multicast addresses are never subject to Neighbor Discovery. These two features make them special enough for an attempt to include them in our current topic of on-link determination to make little sense.

In cases 1 and 2 the on-link address ranges are already represented by prefixes. Case 3 isn't as common but even in it such a range can be reduced to a set of prefixes. Therefore, instead of some arbitrary address ranges, we can speak of on-link prefixes. If the list of its on-link prefixes could be communicated to the IPv6 host, the latter would be able to optimize traffic to its direct neighbors and to turn to its default router less often. The corresponding data structure in the IPv6 host memory has an obvious name: **Prefix List**. If the destination address contains one of those prefixes, it is on-link for the local host's packet sending purposes.

A remark on terminology may be due here. A prefix can be *contained* in an IP address whereas the address can *belong* to the block of addresses represented by that prefix. Therefore it wouldn't be right to say that an address belongs to a prefix, contrary to what our editor maintained, because that would be a distortion of the relations between these entities.

The experience we have just received suggests that care should be taken when populating the Prefix List. The challenge is to not overestimate the reachability of destinations in case the link topology proves non-trivial. So initially there is just one item in the Prefix List: the link-local prefix *FE80::/10* [139, Section 5.1][151, Section 3].

There is a side effect from the link-local prefix being always treated as on-link: The IPv6 hosts of an NMBA link will be unable to communicate using link-local addresses most of the time, unless there happens to be a shortcut connection. This quirk has to be tolerated because of the special role link-local addresses play. Although the IPv6 scoped address architecture from Section 2.4 permits link-local destinations to be routed back into the same link, this feature shouldn't be relied upon by default.

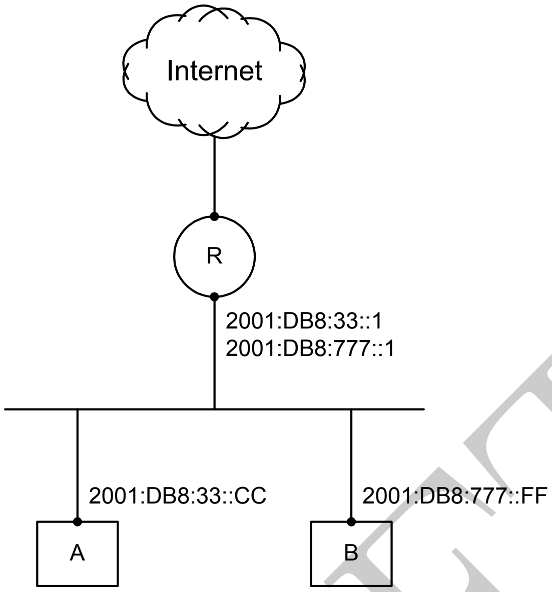


Figure 5.18: One link, one router, two subnets

At the same time, the link topology can't be inferred from the interface hardware type. For example, an *Ethernet* LAN can be reduced to NBMA topology using Private VLAN. For this reason, the IPv6 host must not add a subnet prefix to the Prefix List only because it is assigned to a local interface of a particular technology.

By our master plan, the detailed link topology info will be made available to the router. So the router will be in the best position to be providing the hosts of the link with updates to their Prefix Lists. Although we have no suitable mechanism for that yet, it will be born soon, in Section 5.4.2.

On the other hand, the connected subnet prefixes no longer affect which destinations are considered on-link or off-link by the IPv6 host, contrary to what we were accustomed to in IPv4. Once the subnetting and the destination taxonomy are decoupled in IPv6, it should be perfectly possible for a destination to appear on-link with respect to the host without having to belong to the host's subnet. We will now consider this point using the example configuration from Fig. 5.18, where two distinct subnets are assigned to one link.

How would an equivalent configuration work in IPv4? Suppose host A had the address *192.0.2.33/24* on its interface, host B had *203.0.113.44/24*, and router R had *192.0.2.1/24* and *203.0.113.1/24*. In that configuration, all traffic between A and B would have to be forwarded by R notwithstanding a shared link. Why so? Even if host A had received an ICMPv4 Host Redirect

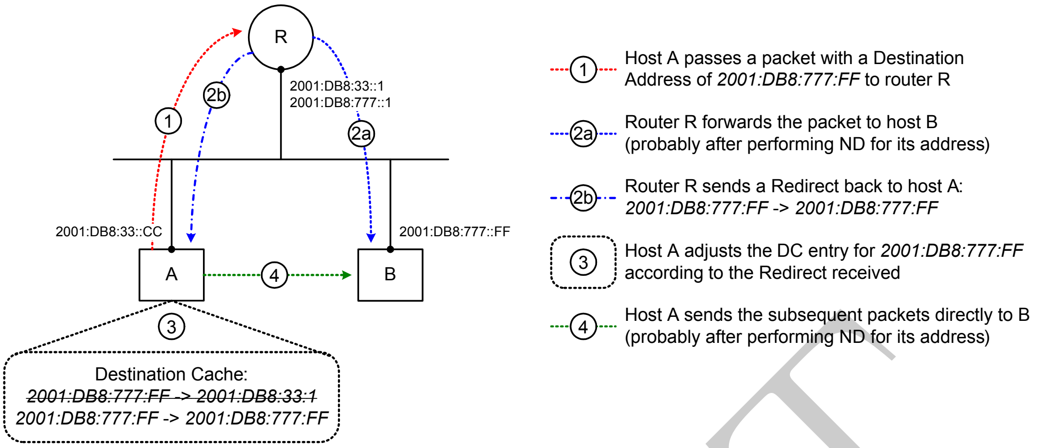

R
2001:DB8:33::1
2001:DB8:777::1
1
2b
2a
2001:DB8:33::CC
2001:DB8:777::FF
A
4
B
3
Destination Cache:
~~2001:DB8:777:FF -> 2001:DB8:33:1~~
2001:DB8:777:FF -> 2001:DB8:777:FF
1 Host A passes a packet with a Destination Address of 2001:DB8:777:FF to router R
2a Router R forwards the packet to host B (probably after performing ND for its address)
2b Router R sends a Redirect back to host A: 2001:DB8:777:FF -> 2001:DB8:777:FF
3 Host A adjusts the DC entry for 2001:DB8:777:FF according to the Redirect received
4 Host A sends the subsequent packets directly to B (probably after performing ND for its address)

Figure 5.19: IPv6 Redirect at work

of the basic form *203.0.113.44* → *203.0.113.44*, it would have disregarded it since its next hop wasn't directly connected [51, Section 3.2.2.2]: *203.0.113.44* didn't come from the subnet *192.0.2.0/24*. In its turn, router *R* wouldn't bother to send such a Redirect to host *A* in the first place [84, Section 5.2.7.2] because it would have been ignored by host *A* anyway. The return traffic from *B* to *A* would be governed by the same considerations with just the host addresses swapped.

The new reality of IPv6 is such that host *A* from Fig. 5.18 will have to accept a "*B* → *B*" Redirect from router *R* simply because the router knows better since it is the authoritative source of such information about the link topology [139, Section 8.1]. In its turn, router *R* should no more restrict its Redirects to one subnet if it is aware that hosts *A* and *B* are neighbors. Then host *A* will be able to learn that *B* is on-link and make good use of that information as shown in Fig. 5.19.

Please make sure you have fully grasped this important difference between IPv4 and IPv6 Redirects [139, Section 3.1].

It will be natural to extend this feature to the Prefix List as well. Then, to be included in that list and regarded as on-link by the IPv6 host, a prefix won't need come from a subnet the host interface is in. For example, if host *H* has got the address *2001:DB8:33::33/64*, its Prefix List can still include the prefixes *2001:DB8:33::/48* (one shorter than the subnet prefix) and *2001:DB8:777::/64* (a different subnet). All IPv6 addresses with either prefix in them will be on-link with respect to host *H* and the latter should be able to communicate with their owners directly, at link layer.

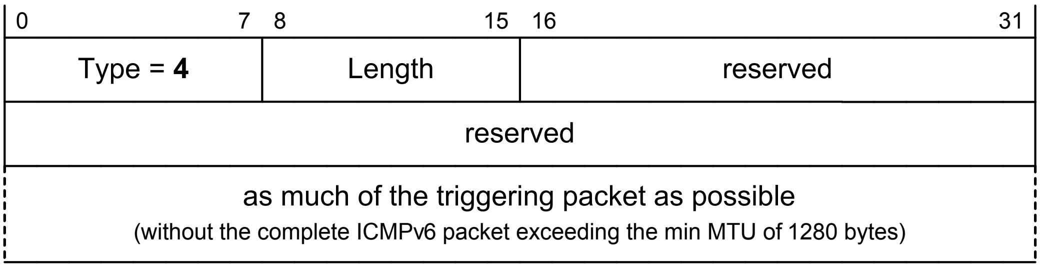

0
7
8
15
16
31
Type = 4
Length
reserved
reserved
as much of the triggering packet as possible
(without the complete ICMPv6 packet exceeding the min MTU of 1280 bytes)

Figure 5.20: Redirected Header option format

Now it is time we worked out the Redirect message format. Owing to its role, it nicely fits in the ND subset of ICMPv6 [139, Section 4.5] as planned. Still, it has an interesting combination of features not found in the other ND messages. On the one hand, the Redirect is a useful hint to the host rather than an error notification and so its well-known ICMPv6 **Type** value of 137 rightfully comes from the Informational Messages range, 128–255. On the other hand, the Redirect is triggered by an arbitrary IPv6 packet instead of a specific protocol query, and this aspect makes it akin to ICMPv6 error messages. As we discussed in Section 4.3, in this case the control message needs to include as much of the triggering packet as possible for its recipient to handle the control message in the proper context of the transport or application session the original packet belonged to.

These seemingly conflicting requirements are quite easy to meet thanks to ND options being available to the Redirect. Instead of having to stick to the ICMPv6 error message format from Fig. 4.8, the Redirect can include the triggering packet, or just its start if it was large, wrapped in an ND option designed for the purpose as shown in Fig. 5.20. The option name will be **Redirected Header**, and its semantics can be as straightforward as this:

- The Redirected Header option makes sense only in a Redirect message. If found in an ND message of some other type, it must be ignored.

- The Redirected Header option data is as much of the triggering packet as possible without the resulting ND/ICMPv6/IPv6 packet exceeding the minimal MTU of 1280 bytes, so as to prevent a Redirect from ever triggering a Packet Too Big condition.

And what will the fixed fields of the Redirect message body be? A Redirect message is just to tell that the best path to destination B is via next hop N, where B and N are both IPv6 addresses. So, apart from some alignment padding, the fixed part of the Redirect message body can consist of

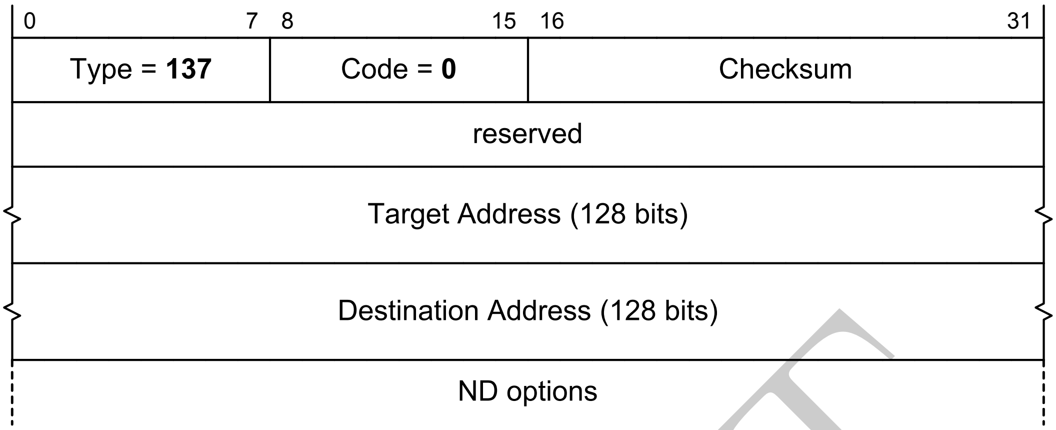


Figure 5.21: Redirect message format

just this pair of IPv6 addresses as shown in Fig. 5.21. First comes the next hop address N. Since its owner is supposed to be the Redirect recipient's neighbor, in the ND lingo this field will be referred to as **Target Address**. Immediately following it is **Destination Address**.

Target Address is placed before **Destination Address** for the Redirect message to have a format similar to that of other ND messages, where **Target Address** is found at the same spot.

Granted, the same destination address can be fetched from the IPv6 header of the triggering packet included in a Redirected Header option, but duplicating it in the fixed part of the Redirect will make its recipient's job a tad easier.

Now yet another difference between IPv4 and IPv6 Redirects can be noticed. As we might still remember, an IPv4 Redirect could be for a single host or for a network, although its format was classful and never accommodated CIDR. At the same time, an IPv6 Redirect can only be for one host so that its information can be retained in the Destination Cache.

The addresses B and N found in a Redirect can be the same. This is the case of interest to us now because such a Redirect is sent by a router to indicate that node B, aka N, is on-link.

The case where the addresses B and N are different may not be as interesting to us at the moment but it has clear practical value. Such a Redirect indicates that the best path to destination B is via a different router on the same link, namely, N.

The other requirements to a valid Redirect message are quite predictable [139, Sections 8.1 and 8.2]:

- The Redirect mechanism must be safeguarded with GTSM (see Section 5.1) from rogue Redirect messages injected from off-link. So the IPv6 **Hop Limit** value must never be less than 255 in a valid Redirect.
- The IPv6 **Destination Address** of a Redirect packet must not be multicast because that would be a clear sign of an attack.

> This requirement isn't found in the RFC text but is really easy to prove. The IPv6 Destination Address of a Redirect packet is always the same as the IPv6 Source Address of the triggering IPv6 packet [139, Section 8.2], and the latter address must never be multicast, as was said in Section 3.2.

- The **Destination Address** B in the Redirect body must not be multicast because Redirects are never triggered by multicast packets.
- The router issuing a Redirect packet must verify that its IPv6 **Destination Address** is on-link and not elsewhere. In other words, the IPv6 **Source Address** of the triggering packet must be on-link in the first place for the Redirect to make sense.
- The recipient of a Redirect message will use its Destination Cache to ensure that the IPv6 **Source Address** of the Redirect packet is the same as the cached next hop for the **Destination Address** B shown in the Redirect body. Indeed, a legitimate Redirect can be expected only from the router that is currently used by this host to reach destination B.

> This check also protects from Redirects duplicated and/or delayed by the network. For example, if host A was redirected by router $R1$ to router $R2$ and by the latter to router $R3$, a delayed copy of $R1$'s Redirect received afterwards will be just ignored by host A.

> And what if the Redirect comes from the current default router but no existing DC entry is found for the redirected destination B? Although the ND standard considers such a Redirect valid [139, Section 8.1] and recommends the DC entry be created [139, Section 8.3], such a Redirect can't but be suspicious because it is about a destination the host has recently sent no packets to. To make a better sense of this situation, the host can consult the copy of the original packet coming in a Redirected Header option.

- The neighboring IPv6 routers are supposed to be known to each other as well as to the hosts of the link by their link-local addresses [139, Section 8], as we mentioned in Section 2.5. Consequently, the following rules need to be enforced:
  - The IPv6 **Source Address** of a Redirect packet must be link-local because the Redirect can be originated only by a router of this link.
  - In the Redirect body, the **Target Address** N must be either link-local (redirecting to an intermediate router) or the same as the **Destination Address** B (redirecting to the destination host).

> The same trend is exhibited by the IPv6 routing protocols. For example, the IS-IS [152, Section 3] and OSPF [52, Section 2.5] adjacencies are to be established between IPv6 routers using their link-local addresses.

> As long as IPv6 hosts know the neighboring routers by their link-local addresses, the hosts won't have to exclude the routers from the on-link/off-link classification, the link-local prefix *FE80::/10* being automatically on-link and inserted into the Prefix List. Otherwise a host would have to consider its default router on-link as an exception from the more general criteria based on the Prefix List and the Destination Cache. That wouldn't be too big a problem, but the conceptual packet sending algorithm would be made less streamlined.

This Redirect format already provides for notifying a host that destination such-and-such is on-link. But will it be sufficient in all cases? Suppose host A sent a packet destined for B via router R, got a "$B \to B$" Redirect back from R, and saved that mapping in its Destination Cache. What will host A be doing next? As soon as there is another packet for destination B, the DC entry will indicate that B is on-link, but B's link-layer address still won't be known and so it will be neighbor discovery time. The success of the subsequent ND exchange will depend on B's ability to receive multicast packets from A because the opening Neighbor Solicitation will be sent to the solicited-node multicast address derived from address B as was discussed in Section 5.1.

What if the link in question is NBMA without multicast capability between its leaf stations, i.e., hosts? In that case host A would still be able to unicast to neighbor B if only B's link-layer address were somehow made known to A. The multicast Neighbor Solicitation won't reach B over such a link though, precluding the normal way of learning B's link-layer address by means of regular ND from Section 5.1.

This problem as posed by NBMA can still be overcome if the new function of the IPv6 router is given a final touch. As we said, the IPv6 router will effectively guide the hosts of the link in their packet exchange. Then it will be up to the router to let host A know all information necessary. In particular, to solve our immediate problem, the Redirect message as sent back by the router just needs to be supplemented with a TLLA option indicating B's link-layer address.

The inclusion of a TLLA option in a Redirect can also be useful over a broadcast link because that will save host A a round of ND, enabling it to send packets to B as soon as the Redirect comes through. Of course, what is a mere optimization over a broadcast link is going to be vital to host-to-host communication over NBMA links, making them usable by IPv6 without the hassle of having to adapt them to broadcast semantics first.

What is host A supposed to do with the link-layer address of neighbor B learned from the TLLA option? Apparently, this data is to be put in the Neighbor Cache. The information coming from a third party, the router, the mutual reachability of A and B is still unconfirmed and the NC entry has to be born in *STALE* state according to the ND rules from Section 5.1. It isn't an issue though because such an entry is immediately usable for packet sending. As soon as neighbor B responds with application data, its NC entry will be promoted to *REACHABLE* state on an upper-layer signal. And if the application protocol happens to be unidirectional and imply no return traffic from B to A, host B will anyway have to respond to a *unicast* Neighbor Solicitation from host A as soon as the NC entry hits *PROBE* state. That Neighbor Solicitation will need to include an SLLA option for host B to be able to respond to it under such conditions.

Note that all of the rules we have discussed apply to host B as well: It must not consider node A on-link and try to send application packets directly until redirected by the router.

The last question regarding this mechanism is how router R can learn the link-layer address of host B in the first place when no real multicast is available at link level. Of course, the means of last resort suiting any NBMA link will be to load R's Neighbor Cache beforehand with static data. At the same time, router R will be able to auto-discover neighbor B provided that the NBMA link itself or the adaptation layer sitting between the link layer and the IPv6 stack can emulate a limited form of multicast, so-called "pseudobroadcast" [157], in which multicast packets sent by router R are fanned out to all hosts of the link while multicast packets from each host simply go back to router R—think of point-to-multipoint star topology. In this case router R can use regular ND to resolve B's address and will certainly do so when forwarding the initial packet from A to B. In addition, B's link-layer address may already be cached by R once B discovered its default router R, e.g., in order to send a packet to an off-link destination. If that happened, the multicast Neighbor Solicitation from B contained an SLLA option with B's link-layer address, thus permitting R to create an NC entry for B in *STALE* state.

Up until now we have deliberately limited ourselves to the simplest scenario where the prototype link has just one router on it while each host has just one network interface besides the loopback. How difficult will it be to get these limitations lifted now? Let's begin with the former and consider a link with multiple IPv6 routers on it. For an IPv6 host connected to that link to benefit from the variety, its single default router address setting needs to be replaced with a whole list of default router addresses. Thus a new IPv6 host data structure is born: the **Default Router List**. For the moment, assume this list is static, e.g., loaded from a local configuration file; its automatic population is left until Section 5.4.2. For simplicity, the precedence of the routers so listed will be just the same.

Unable to prefer one router over the rest, an IPv6 host will start its network access by selecting a random entry from its Default Router List. This will be sufficient, e.g., to distribute the load from a densely populated link over multiple routers. Besides, router redundancy can be a way to fault tolerance if each host can detect that its current default router has gone AWOL and fail over to another entry in its Default Router List. Will a new protocol have to be invented for that? Fortunately, no it won't, since we already have the wonderful mechanism of NUD from Section 5.1 requiring nothing but the basic ND protocol and supported by every IPv6 node out there. If the NUD process signals unreachability of the current router, the host will try the next router in the list and so on. Of course, an unreachable router shouldn't be completely dropped from the Default Router List because chances are that by the time the last default router in the list fails, the first one will be back up. So the Default Router List just needs to be rotated instead. That's all an IPv6 host will need in order to start effective use of multiple routers.

> No doubt, finding a live router is open to optimization. For example, the host can be continuously monitoring the status of all its default routers with NUD and ranking them by the NC entry state as follows: *REACHABLE* is better than *STALE* and *STALE* is still better than *INCOMPLETE*.

> As a matter of fact, the reference IPv4 host was also supposed to support multiple default routers [51, Section 3.3.1.2]. At the same time, it had to be admitted that IPv4 lacked a well-known reliable mechanism to detect router unreachability [51, Section 3.3.1.4]. This conflict between the theory and the practice couldn't but result in very few IPv4 host implementations supporting a list of default routers.

Can we offer a solution of equal elegance to the second problem, that regarding multiple host interfaces? To start with, let's recap the basic truth that neighbor relation is defined only within a given link. In other words, node B can't be node A's neighbor "at large": They will be neighbors, if at all, on a certain link L. In turn, a host's window into the link is its network interface while the IPv6 host data structures introduced so far are tied closely with the notion of neighborship:

- A Neighbor Cache entry contains information about a neighbor, mapping its IPv6 address to its link-layer address.
- A Destination Cache entry resolves to a next hop, which is to be a neighbor.
- A Default Router List entry must be a neighbor to be usable by the host.
- An on-link prefix from the Prefix List can be present only in a neighbor's IPv6 address.

Therefore it is no surprise that these data structures have to be bound to a specific network interface as they will make no sense in the context of another interface.[10] As long as the host has just one interface in addition to the loopback, this binding can remain implicit; but as soon as the host acquires more interfaces, a separate instance of each data structure will have to be maintained on each interface, while there is no master data structure to rule them all. The bare-bones IPv6 host thus is given no criteria to prefer a particular interface for sending a given packet, that decision left up to more specialized mechanisms and protocols [139, Appendix A].

For example, an advanced host implementation with multiple interfaces can participate in a routing protocol and calculate the best routes itself instead of having to rely on the one-size-fits-all concept of default routers.

It may seem that there are cases where our reference host could still make the best choice of egress interface. A typical fallacy would go like, "If there is an interface where the destination is on-link, it should be preferred." But what if that link has silently failed but the destination is still reachable via a router on a different link?

The point is, multiple connections are employed to solve quite complex problems such as high availability where simple solutions just aren't going to cut it. At the same time, the basic mechanisms of the IPv6 host need to stay simple enough to be implementable in a wristwatch, let alone a toaster or coffee-maker.

As far as our goals are concerned, it will have to be assumed that *the reference IPv6 host has a default interface all packet sending is done through.*

Now we should bring all points of our discourse together in order to work out a conceptual sending algorithm for the reference IPv6 host [139, Section 5.2]. This algorithm will be invoked on each outbound packet. The input to it will consist of the IPv6 Destination Address of the packet to be sent and the intended egress interface, and the output from it needs to be the link-layer address of the next hop, or an error indication if the lookup failed.

Strictly speaking, the first branch has to be on whether the Destination Address is multicast or unicast. This decision will be really easy to make as the IPv6 multicast addresses have a distinctive prefix in them, *FF00::/8* (see Section 2.3). The further handling of a multicast destination address isn't relevant to us right now because, by our plan, any IPv6 multicast destination

[10] Unless the other interface is connected to the same link.

is treated as on-link and the link-layer multicast destination address can simply be *calculated* from its IPv6 counterpart using the procedure defined along with the rules of how IPv6 is to be encapsulated over this particular link type. For example, *Ethernet* employs the trivial operation we are already familiar with, in which the well-known MAC prefix `33-33` is prepended to 32 low-order bits of the IPv6 multicast address. So let's now focus on the more complex case of a unicast destination.

In the next step, the sending host needs to select the IPv6 next hop address for the outbound packet. To save time, it will employ the Destination Cache of the egress interface requested. If the Destination Address is already cached, being the lookup key of an existing DC entry, that entry's RHS value will reveal as many as two pieces of information: Explicitly it will tell the IPv6 next hop address while implicitly it will indicate if the destination is on-link. However, the latter won't be too important to the host once it knows the former.

On the other hand, if the Destination Address isn't cached yet, the host will have to determine whether the destination is on-link. The information available to the host may be incomplete, and so it will be safer to err on the off-link side because then the packet will be sent OK via a default router. By contrast, erring on the on-link side will result in a transmission fault if the destination is actually off-link. For this reason, the host tries to match the Destination Address against the Prefix List of the egress interface and considers it on-link only if matched by at least one on-link prefix. Otherwise the destination must be regarded as off-link by the host.

The current ND standard prescribes a longest-prefix match [139, Section 5.2], although such information isn't really necessary for the on-link decision by the host. In fact, should the Prefix List contain a prefix along with its super-prefix containing fewer bits in it, the end result won't depend on which prefix was hit. For example, suppose the Prefix List contains just two prefixes, *2001:DB8::/48* and *2001:DB8::/64.* Note that the former is a super-prefix of the latter. Still, the address *2001:DB8::1234* will be found on-link regardless of which of the two prefixes fired. We would speculate that the idea to use a longest-prefix match has lingered in the ND spec since the times when handling multiple interfaces at ND level was still being researched into. Indeed, then it would be natural for the interface with the most specific on-link prefix to win. However, as we have just discussed, in its current form ND is not concerned with the selection of the egress interface. A more practical reason to do a longest-prefix match on the Prefix List can be to keep track of per-prefix usage stats such as access counters.

Should the Prefix-List test return on-link status for the destination, the IPv6 next hop address will be the same as the Destination Address. Otherwise the sending host will need to pick a candidate router from its Default Router List, e.g., based on the NUD data. The next hop chosen either way will be retained in the Destination Cache with the Destination Address as the lookup key.

> If reachability info is available on none of the default routers, it will be sufficient to scan through the Default Router List entries in a round-robin fashion [139, Section 6.3.6]. This way the current reachability information will soon appear in the Neighbor Cache for at least some of the default routers configured.

This concludes the top half of the IPv6 unicast sending algorithm, whose task was to select the next hop. Now it can be summed up as a flow chart in Fig. 5.22.

Once the IPv6 next hop address is known, the ND module from Section 5.1 will kick in to resolve it. ND also tries to save time and effort by means of a cache, namely, the Neighbor Cache of the egress interface. If the next hop address is already in that cache and its entry is complete, that is, contains a valid link-layer address, the latter can just be returned. Of course, the access to the NC entry can prompt the NUD process to start refreshing it in the background, e.g., if the entry was *STALE*. In the meanwhile, the IPv6 sending algorithm can just return thanks to any complete NC entry being immediately usable for sending purposes, as was discussed in Section 5.1.

Finally, should no NC entry be found for the next hop, the host will have to queue the outbound packet and initiate a Neighbor Discovery procedure. Its end product will be a complete NC entry, provided the next hop is reachable, and the link-layer address from that entry will need to be returned by the sending algorithm.

What are the most prominent features of this algorithm? On the one hand, it is capable of load-sharing across multiple default routers [153] while avoiding the pitfall of random multipathing thanks to caching the next hop per destination.

> The drawbacks of unsystematic multipathing were recapped in Section 5.1.

On the other hand, this algorithm won't bother to check the reachability of the next hop before sending the packet out. Suppose the host has hitherto been using a particular router, but now its favorite router has been abruptly

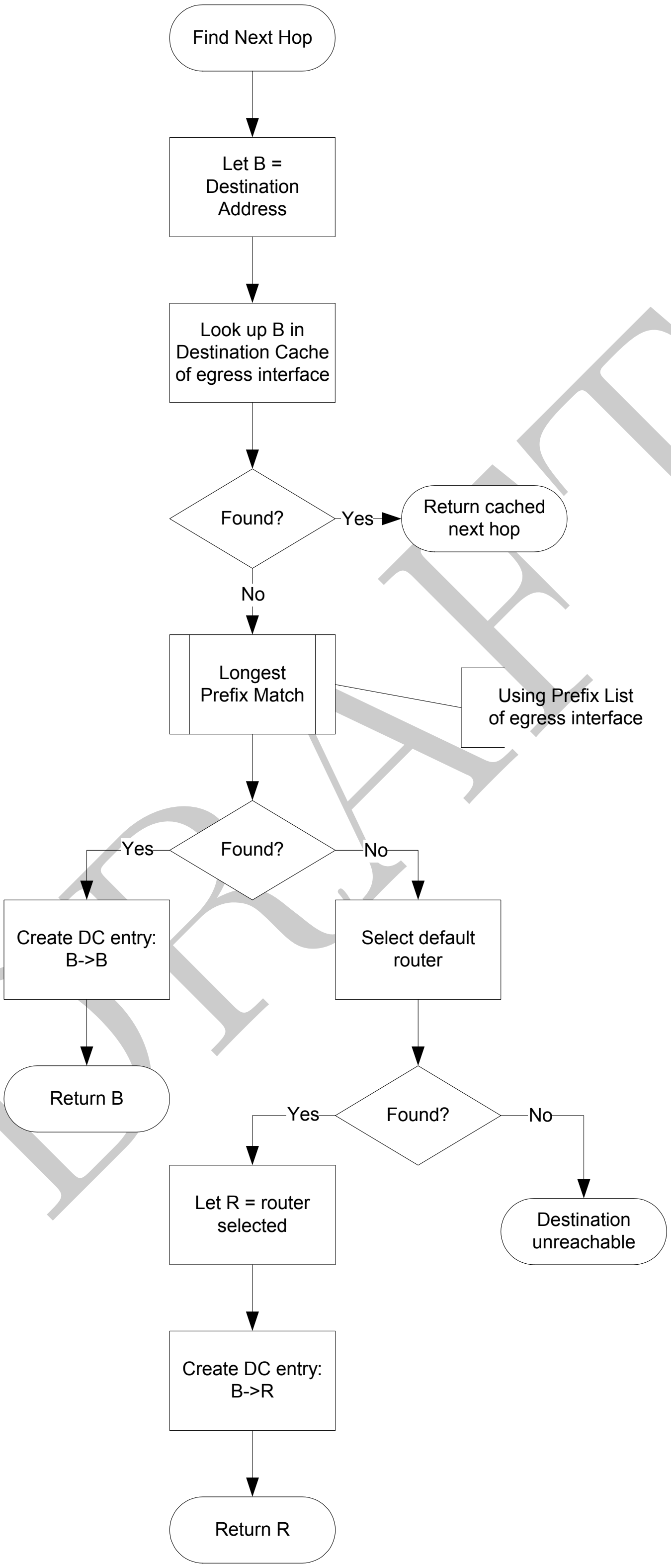


Figure 5.22: Top half of the conceptual sending algorithm: Next hop selection

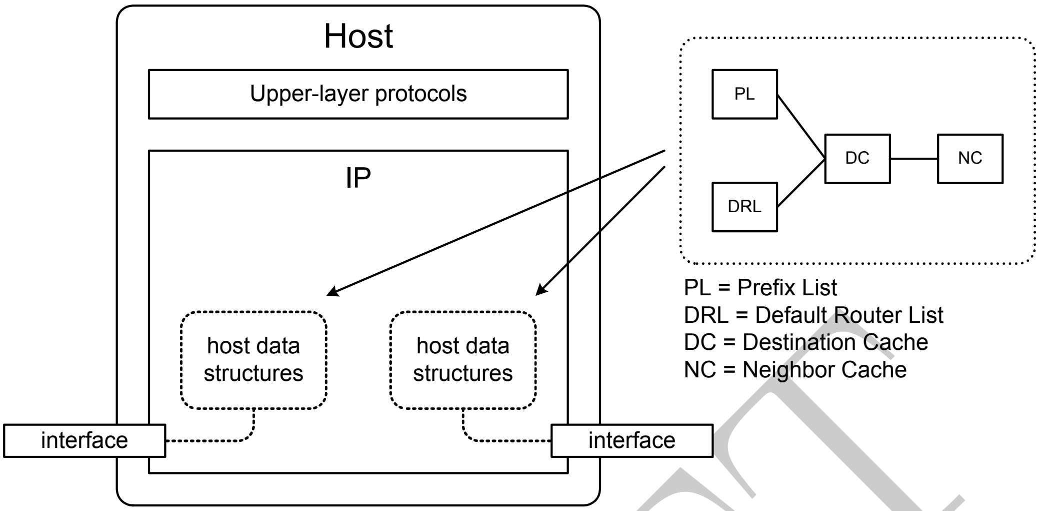


Figure 5.23: Conceptual IPv6 host data structures and links between them

turned off. Then the sending algorithm will keep returning the link-layer address of the unavailable router until the background NUD process signals the router gone and the DC entries resolving to the router's address expire. Only then will the switch-over to a new default router be complete.

Of course, this is a trade-off between reliability and performance: Routers don't usually vanish every second whereas checking next hop reachability prior to sending each packet would result in an unacceptable overhead as well as in a delay roughly equal to the link's RTT. In addition, owing to continuous upper-layer feedback, NUD can use quite short timer intervals and detect unreachability of a neighbor in mere seconds since the last positive signal from an upper-layer protocol.

A natural optimization here will be to drop, along with its NC entry, all of the DC entries resolving to the unreachable neighbor. This step will prompt the local host to find a new next hop for each destination formerly reachable via the missing neighbor.

Besides, if the neighbor gone unreachable is a default router, it should be temporarily excluded from the candidate routers as considered by the sending algorithm. For instance, if the Default Router List is scanned through in a round-robin manner, it will be sufficient to move on to the next entry. This will effectively place the dead router at the end of the list, giving the poor thing plenty of time to recover before the host has to select it again.

Now it is clear to us that the data structures of the reference IPv6 host are not only bound to a particular interface but also linked with each other as shown in Fig. 5.23.

The forward links are as follows:

- The host uses the Default Router List and the Prefix List to work out the next hop for an outgoing packet and saves the result in a Destination Cache entry.
- The Destination Cache entry resolves the Destination Address to the IPv6 next hop address.
- Next, the host carries out Neighbor Discovery using the next hop address as a target address and saves the result in the Neighbor Cache.
- The end result is that the host is able to send the outgoing packet to a specific link-layer address.

The feedback links are as important; they are as follows:

- In the background, the NUD process is monitoring the status of the neighbors having entries in the Neighbor Cache.
- If unreachability of a neighbor is detected, all of the DC entries where that neighbor is the next hop are to be removed.

> This step will allow the host to adapt to a router failure as well as to a link topology change when a former neighbor ceases to be on-link and becomes accessible only via a router.

- In addition, if the unreachable neighbor is a default router, the host needs to take its failure into account in the subsequent next-hop decisions for off-link destinations.

> The way for the host to do that has to depend on the Default Router List management algorithm implemented. The conceptual algorithm [139, Section 6.3.6] only recommends that, firstly, the routers known or presumed to be reachable get preference and, secondly, when no such routers are on the list yet, routers be selected in a round-robin fashion. The point is just to defer selecting a failed router again until the other routers have been tried.
>
> With router reachability information supplied by NUD, having no reachable routers left in the Neighbor Cache doesn't mean all of the host's default routers are gone: Gone are only those the host has recently used, and it is time the other Default Router List entries were given a chance. Hence the round-robin selection algorithm.

We have just completed a big job—phew!—and now a quick review is due. In Section 5.1 the IPv6 Neighbor Discovery mechanism was "built" but no criteria were laid down for an IPv6 node to tell when a remote destination was to be considered a neighbor in the first place. Needless to say, looking for a neighbor that wasn't would be as productive as the proverbial hunt for a black cat in a dark room. So immediately afterwards we had to concentrate on neighborship criteria for IPv6. In the course of that, the simple link and subnet model as adopted by IPv4 was put under scrutiny and found too restricted to stand the challenge from the diversity of modern link technologies. To have the historic restrictions lifted, a significantly different model was set down for IPv6, in which the routers would orchestrate the packet sending activity of the hosts over the link while the hosts would learn the link topology from the routers rather than try to work it out from some predefined criteria. Next, a mini-protocol under the ND umbrella was "designed" to support the actual operation of an IPv6 link in keeping with this model.

> To be fair to IPv4, its host, too, was allowed to stick by a similar model where it would have a default router list and a route cache updated with ICMP Redirects instead of a full-fledged routing table [51, Section 3.3.1.2]. At the same time, the IPv4 neighborship criteria were unambiguous in that all connected subnet addresses, and only they, were neighbors' [51, Section 3.3.1.1].

How original do you think our model is? Well, they are right to say nothing is new under the moon, for essentially the same model has long been relied upon by the ISO stack. In particular, the interaction of its hosts (aka End Systems, ES) and routers (aka Intermediate Systems, IS) is governed by a specialized protocol known as ES-IS [154] (an earlier version available as [155]).

The ES-IS way can even be regarded as more elegant and comprehensive. Not limited to mere redirects, it involves periodic advertisements the ES and IS make to let the other party know of their presence. Thanks to that, a form of neighbor discovery is needed in ISO only when there is no IS on the link. Even in that case the source ES will waste no time on unicast address resolution, sending the data packet to the All ES link-layer multicast address instead. The destination ES, if actually present on the link, will accept the packet based on its ISO unicast destination address; at the same time it will send the source ES an advertisement with its link-layer address. Upon the reception of the advertisement, the source ES will be able to cache the neighbor's link-layer address and start unicasting the subsequent packets for the same destination.

The ISO ES thus regards all destinations as on-link by default. By contrast, the IPv6 host must assume by default that all destinations except the link-local ones are off-link. Unless provided with a router or a manually populated Prefix List, IPv6 hosts can do unicast communication only using their link-local addresses because under these conditions all of the other IPv6 unicast addresses are off-link and hence unreachable.

In fact, the earlier IPv6 standards leaned towards the ES-IS model where all destinations were on-link by default, but it had to be turned down [151] once the subsequent experience brought it to light that there were serious problems lurking behind its apparent simplicity [156].

If you would like to get a quick introduction to the ISO stack and in particular the ES-IS protocol, the wonderful book by Radia Perlman [118] can't be recommended highly enough.

To conclude this section, let's consider a practical case highlighting the differences between subnet management in IPv4 and IPv6. As we may remember, IPv4 allowed for a longer prefix to be taken from a subnet and assigned to a link elsewhere. Suppose the prefix *192.0.2.0/24* was assigned to link $L1$ and the prefix *192.0.2.96/27*, to link $L2$ as shown in Fig. 5.24.

Routers $R1$ and $R2$ would have no problem handling that configuration because the $L2$ prefix was more specific than the $L1$ prefix, enabling unambiguous routing of the IPv4 range *192.0.2.96–192.0.2.127* towards link $L2$ and the rest of *192.0.2.X* addresses towards link $L1$. Nevertheless, hosts on link $L1$ would fail to communicate with their peers on link $L2$ because the former would consider the latter directly connected and so try to discover them through ARP instead of sending the packets via router $R1$. The remedy for this problem would be either to enable ARP proxy on $R1$ or to let

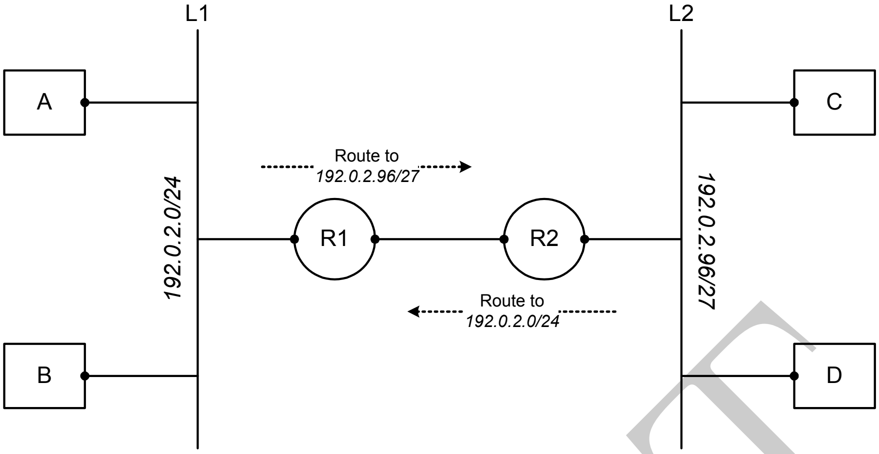


Figure 5.24: Remote subprefix in IPv4

the $L1$ hosts know the more specific route to *192.0.2.96/27* via $R1$ by means of their static configuration or a routing protocol.

Now consider an equivalent IPv6 scenario as shown in Fig. 5.25. Suppose link $L1$ has the subnet prefix *2001:DB8:1:2::/64* and link $L2$ has *2001:DB8:1:2:3:4::/96.* First of all, we can't help pointing out that such subnetting will be at odds with the IPv6 unicast addressing policy discussed in Section 2.6 because it involves a subnet prefix whose length isn't 64 bits. At the same time, such a subnet prefix isn't in violation of the core protocol, so its use will be left to the network admin's judgment.

As for the faultless operation of such a setup, there are at least three ways to ensure it using nothing but the basic IPv6 mechanisms:

- One option is to leave the Prefix Lists of the $L1$ hosts empty except for the link-local prefix, thus forcing those hosts to send all of their packets via $R1$ at first for $R1$ to redirect them back to the link as necessary.
- Another option will be to flag the entire *2001:DB8:1:2::/64* as on-link for the $L1$ hosts and enable ND proxy for *2001:DB8:1:2:3:4::/96* on $R1$.
- Lastly, a compound Prefix List can be loaded on the $L1$ hosts that is equivalent to *2001:DB8:1:2::/64* less *2001:DB8:1:2:3:4::/96.*

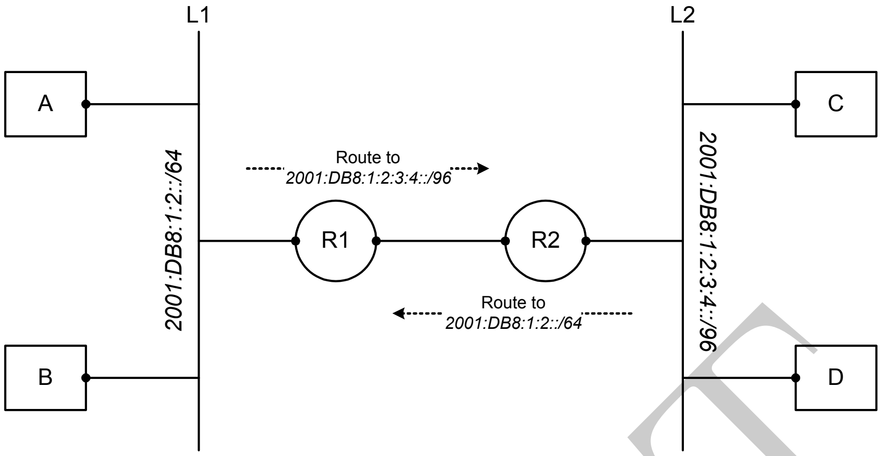


Figure 5.25: Remote subprefix in IPv6

As an exercise, the reader will find a way to build such a Prefix List with as few entries as possible in it. (Hint: *Start by proving that the shortest solution possible has 32 entries in it. Then try to prove that the shortest solution is unique. If you do that, you should be able to see what the list will look like.*)

5.3 IPv6 Addressing Modes Revised. Anycast

Now that IPv6 Neighbor Discovery has been finalized, we can revisit the addressing modes of IPv6. In particular, we are dying to see what has become of the good old broadcast. Up to this point, this topic hasn't been raised in full because we weren't ready to make a final decision on it; and now we can tell with confidence that IPv6 can go without broadcast thanks to its reliance on, and mandatory support for, multicast. IPv4 broadcast was based on its link-layer counterpart, and we regard the latter as outmoded because of its inefficiency with respect to the link resources: Each broadcast frame has to be forwarded to each station of the link notwithstanding its Layer 3 protocol preferences. If need be, all IPv6 nodes within a zone can be addressed using the suitably scoped All Nodes multicast address, *FF0x::1*. The bottom line is that *IPv6 has no broadcast because it has been completely superseded by multicast.*

Having no broadcast renders special broadcast addresses unnecessary in IPv6. In particular, the interface ID with all bits set to one, *FFFF:FFFF:FFFF:FFFF*, is not reserved, unlike in IPv4.

> That said, such an interface ID still can't just be used in IPv6 addresses supposed to have a Modified EUI-64 in them. To see why, check out the values of the **U/L** and **I/G** bits in the interface ID of all ones.

Multicast can be of use not only to control protocols but also to application ones. It comes handy when a single packet needs to be fanned out to multiple recipients with a minimal overhead. For example, an NTP server can be multicasting time signals to a group of NTP clients while a DHCP client can multicast its request to a whole group of DHCP servers, receive several offers, and pick the best one.

However, there are apparently similar tasks multicast still can't do. For instance, it would make little sense to set up a multicast group of DNS servers at least because the DNS protocol was designed with unicast in mind and a DNS client can't really benefit from multiple responses coming through.

> The reader may argue here that DNS plus multicast makes mDNS. However, the latter protocol serves quite a different purpose: It provides for peer-to-peer name resolution in a serverless LAN.

Neither can, for obvious reasons, a TCP-based protocol benefit from multicast. But even so the concept of an interface group sharing the same IPv6 address remains attractive where multicast can't be applied. E.g., in a large network all DNS or SMTP servers would be assigned the same magic address X so that a client could use just one software configuration to roam the network and still speak with the nearest server at each spot as shown in Fig. 5.26. From the client's viewpoint, its packet would hit a random member of group X, so this addressing mode is known as **anycast**.

Behind the scenes, however, anycast can't be as random as it may seem to the packet source. Granted, stateless protocols over UDP won't care which server is going to be hit with each request, but TCP does not throw dice. For that matter, any stateful protocol requires all packets of one session go to the same anycast node at least as long as the network configuration remains constant. The conclusion from that sounds more like an oxymoron: Anycast must be reproducible.

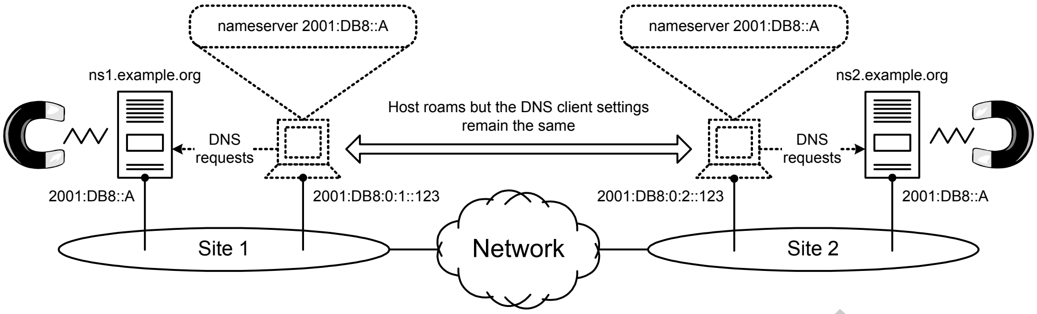


Figure 5.26: Anycast DNS: the raw idea

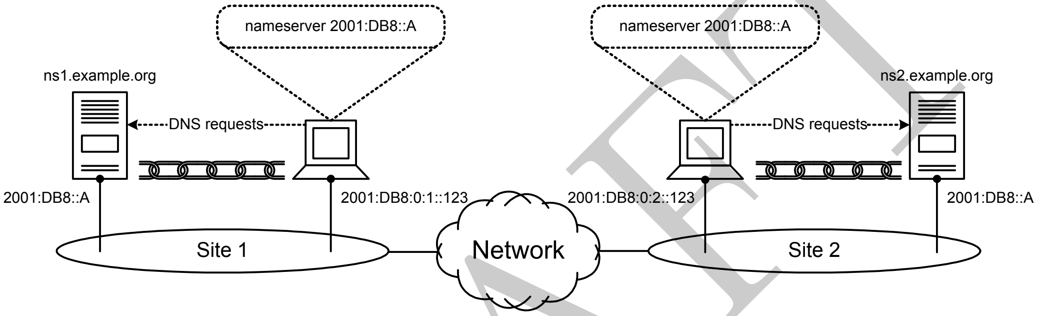


Figure 5.27: Anycast must be reproducible

DNS could have been a textbook example of a stateless protocol if not for its use of virtual circuits over TCP as an alternative to the simple UDP transport [158]. In particular, TCP has to be used if a DNS message is overlong. So in actual fact even DNS requires anycast be reproducible, as illustrated by Fig. 5.27.

Even a stateless protocol will require anycast be reproducible if the length of its message when in a complete IP packet can exceed the minimal MTU value. This problem concerns, of course, fragmentation and reassembly. Once an IP packet is fragmented, each of its fragments will travel to the destination independently of the rest, but all of them need to come together at the same node for their reassembly to be possible.

How can these conflicting conditions be met, if at all? To get some real food for our thought, let's first consider two isolated cases at the opposite ends of the spectrum. One case is where the source and the anycast destinations are spread out over the network so that they aren't neighbors. The other

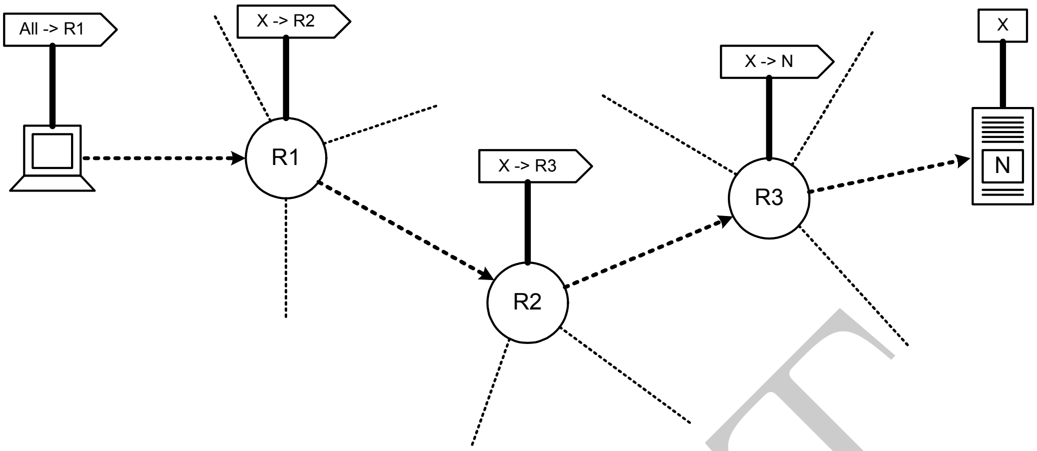


Figure 5.28: The path to an anycast destination

case is where the source and the anycast destinations all are on the same link.

The first case is solvable by means of mere routing configuration. Suppose the source is a basic IPv6 host having just a default router. By the assumption, the anycast destinations aren't its neighbors and so the anycast packet will have the router as its first hop. Provided the router has a route for the anycast address X pointing towards the nearest anycast node N, in its next hop the packet will either reach its destination or get a hop closer to it, and so on. In a nutshell, it will be sufficient to configure a chain of IPv6 routes leading the packet through the routers from its entry point to the closest anycast node with address X as shown in Fig. 5.28. Conceptually, this scheme won't be too different from conventional routing in a large meshed network, where multiple paths exist and it is normal for packets originated at different spots to take different paths to the same destination. So the tools to implement it will be just the same: either static routes or a dynamic routing protocol.

> And what if a router has multiple equal-cost routes to the anycast address X? In that case its task will be no different than in unicast ECMP routing: to ensure that all packets of one flow, for any suitable definition thereof, use the same route. As a byproduct, the packets of one flow will be hitting the same anycast node in the end—too easy!

Yet unclear is only the ultimate hop where the last router delivers the packet to the anycast node. There are at least two ways to handle this step.

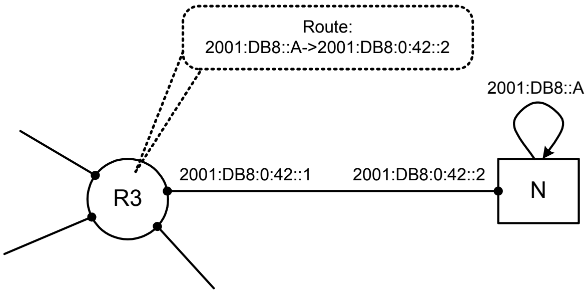


Figure 5.29: Anycast: the last hop handled through routing

One option is to make the router believe that address X is on-link. How to do this can be implementation-specific, e.g., the subnet $X/64$ can be assigned to the link connecting the router to the anycast node. In any case, the router will do Neighbor Discovery on the target address X and the anycast node will respond and get the anycast packet.

Another option is to assign a regular unicast address A_J to each anycast node J, in addition to the anycast address X. Then, for the last router to forward the anycast packet to the specific anycast node N, the following route will be sufficient in its forwarding table: $X \rightarrow A_N$. In this scheme addresses A_J can be link-local, and having such an address is a must for any IPv6 interface anyway, as was discussed in Section 2.5. As for the anycast address X, in this case it can appear on any interface of node N including the loopback, as illustrated by Fig. 5.29.

When the source host has to move to a different spot in the network, it will get a new default router there to suit its new connection, and that will be the beginning of a different chain of routes leading to a probably different anycast node M, as illustrated in Fig. 5.30. Because the chain of routes used by the source depends only on its current location in the network map, anycast will be reproducible as long as the source stays put, which is a satisfactory solution.

Now on to the second case, where the source and all of the anycast nodes are neighbors on some link. It is hard to tell in this scenario which node is closer to the source, so it will be natural to postulate they all are equidistant and to let anycast be truly random—at first. However, the random choice of an anycast node must be followed by a steady packet exchange with it. Will we manage to solve this problem as well using the tools already available?

Suppose the source is still unaware that address X is special and keeps treating it as a regular unicast address. In the case at issue X is on-link, so the source will start by performing Neighbor Discovery on the target address X. Namely, it will send a Neighbor Solicitation regarding X to the

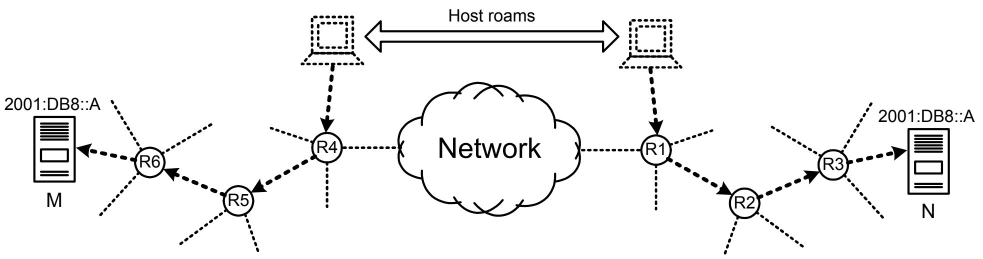


Figure 5.30: The source host speaks to a specific anycast node depending on its connection point

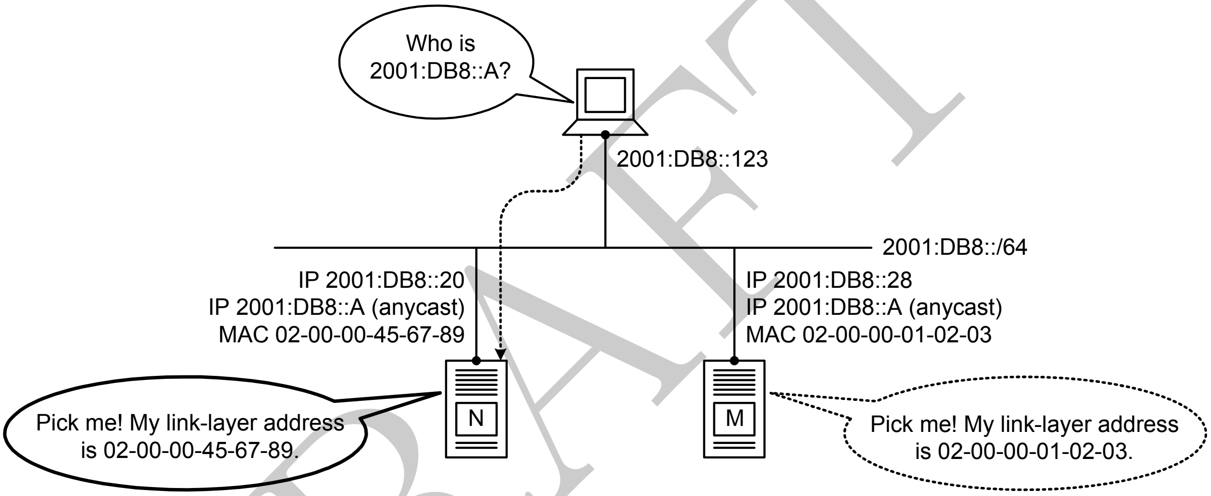


Figure 5.31: Same-link anycast using Neighbor Discovery

corresponding solicited-node multicast address. If the anycast nodes have joined that multicast group, they will receive the NS and be able to respond to it, each with its own link-layer address. Thus the source host is going to receive multiple Neighbor Advertisements with different link-layer addresses in them, as shown in Fig. 5.31.

Potentially, such Neighbor Advertisements can contend for the Neighbor Cache entry of address X, altering it in an unpredictable manner. We therefore need to ensure now that the anycast nodes won't battle for the NC entry. A peaceful outcome will be guaranteed if their Neighbor Advertisements have the **O**-bit set to zero. Then the node whose NA comes first will secure the NC entry. The subsequent NAs from other nodes will be changing the entry's state to *STALE* but it will be immediately recovering back to *REACHABLE* with its cached link-layer address unchanged thanks to NUD. Thus anycast will be made reproducible in this case as well.

Note how similar the mechanisms of same-link anycast from this section and of ND proxy from Section 5.1 are.

Should there be multiple anycast sources on the link sending packets to address X, their sessions will be distributed fairly across all anycast nodes. Thus anycast can be a means for load sharing.

So far our anycast node prototypes respond to a Neighbor Solicitation all together, causing link load spikes. To smooth those out, each anycast node should delay its Neighbor Advertisement for a random time between zero and a specified ceiling[11] [139, Section 7.2.4].

It would make sense for *MAX_ANYCAST_DELAY_TIME* to be shorter than *RETRANS_TIMER* to allow for the link transmission delay, but the current standard [139, Section 10] prescribes equal default values for them.

On the other hand, if the anycast nodes weren't to introduce an artificial delay in their NAs, the source would be able to stick to the closest or least loaded node since the NA to come first wins.

And what if the anycast node currently selected goes down, e.g., crashes or freezes? In that case the NC entry will eventually get removed by NUD and the source host will have to repeat Neighbor Discovery of target X, effectively switching over to a live anycast node, if available. Thus same-link anycast can provide for fault tolerance in addition to load sharing.

The two extreme cases of anycast considered, are we able to have them work together in order to suit all practical scenarios? Suppose there is a router sitting between the source hosts and the anycast nodes, as shown in Fig. 5.32. Then it will be up to the router to select an anycast node, but this won't pose a problem because it is no different from regular ND. However, there is a subtle difference between this hybrid scenario and the same-link one: Should there be multiple sources behind the router, all of them will home in on the same anycast node and there will be no load sharing.

The last question regarding anycast concerns its addressing. Will we need a new address type with "magic" properties? In fact, no: The anycast scheme just "designed" explicitly requires that *the anycast address be indistinguishable from its unicast counterpart*. Only then anycast can build on the essentially unicast mechanisms of routing and ND. So it won't be possible to

[11] Parameter *MAX_ANYCAST_DELAY_TIME*, default 1 second [139, Section 10].

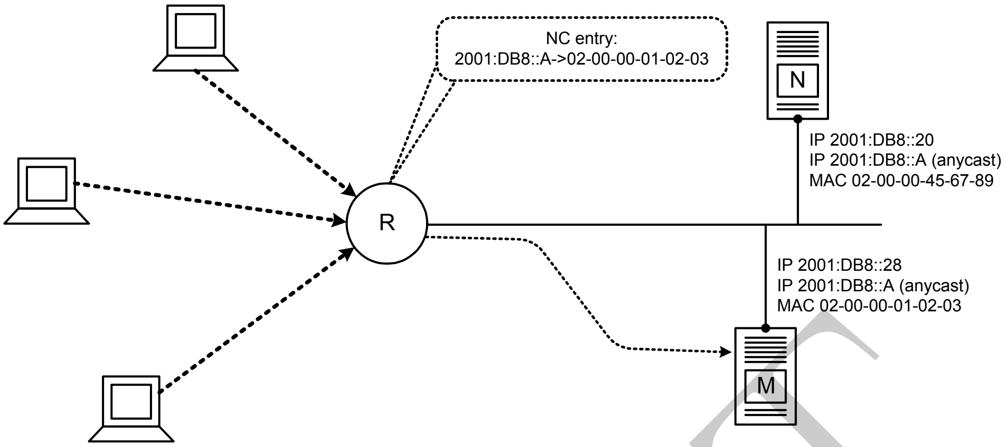


Figure 5.32: Real-world anycast using routing and ND

tell an IPv6 address is anycast unless the configuration of its owner node(s) is examined, and only the owner node(s) will handle the anycast address in a slightly different manner.

This anycast mechanism has no real ties to the IP version, and so it is as possible to implement in IPv4 just using ARP instead of ND [159]. Such anycast has been applied to DNS [160].

There already are well-known IPv6 anycast addresses [161].

Some IPv6 implementations don't support sending packets from an anycast source address [162, 163] albeit such packets can be legitimate if they are sent in response rather than open a new conversation. Their only quirk is that their Source Address can potentially identify more than one interface, but in each particular case this source ambiguity is eliminated by the same mechanisms that make anycast reproducible.

Discuss whether the use of the loopback address *::1* can be regarded as a special case of anycast.

To sum up this section, IPv6 has relinquished broadcast but acquired a new addressing mode instead: anycast. Unicast and multicast are still there, with the latter performing in IPv6 the important roles traditionally played by broadcast.

Unfortunately, the practice shows that the contemporary link-layer multicast implementations, such as those found in *Ethernet* switching components, often lack in maturity, resulting in weird problems inevitably felt at IPv6 level every time link-layer multicast is acting up. Link-layer hardware vendors will have to put some real effort into this area before their gear can qualify for an IPv6 Ready sticker.

5.4 IPv6 Autoconfiguration

5.4.1 Interface ID Selection and Duplicate Address Detection

Once upon a time there was a protocol called IPX and it was somewhat similar to IPv6.[12] An IPX address, too, consisted of two fixed-length parts, one owned by the link and the other by the interface. A feature of the IPX address was that its interface ID part was *always* identical to the MAC-48 address of the interface it was assigned to. Thanks to that, two aspects of IPX operation were made trivial. First, unicast address resolution needed no wire protocol and could be done just locally by extracting the interface ID and using it as the link-layer address. Second, unique addresses didn't need to be assigned per interface: It was sufficient to assign a unique prefix to the entire link and rely on *IEEE* having ensured the uniqueness—global one!—of the MAC address-based interface IDs. The underside of that design was that IPX addresses were linked to the underlying MAC addresses and, for example, replacing the hardware of a network interface would result in its IPX address changed.

If the hardware supported a programmable MAC address, the former IPX address could still be retained after hardware replacement, but that would mean a return to manual address management, nullifying the benefits from the IPX address design. In fact, implementations even allowed to customize the interface ID of an IPX address, but that implied changing the underlying MAC address on the interface.

Of course, another problem with the IPX address design was that IPX could operate only over link technologies whose addressing was compatible with, or mappable to, MAC-48.

[12] A fairer way to tell the same thing may be that it is IPv6 that bears certain similarity to IPX, having borrowed some of its merits.

Fortunately, the IPv6 interface ID is more flexible than its IPX ancestor was. The only significant requirements have so far been that the IPv6 interface ID be 64 bits long[13] (see Section 2.6) and comply with the Modified EUI-64 format (see Section 2.7). In Section 2.7, we even went as far as to alter the original *IEEE* EUI-64 format in order to make locally administered interface IDs more convenient to work with: Owing to the adjustment, they can be small numbers such as 1, 2, 3. User-friendly interface IDs will go a long way toward making static IPv6 addresses manageable, which is a must-have quality as long as there are servers requiring a permanent and, ideally, memorable address.

> For most servers, a permanent DNS name resolving to some IP address will do. However, certain cases call for the server's IP address to be entered into the client-side configuration. One notable example is DNS itself, for trying to learn the nameserver address from DNS would be as effective as pulling oneself out of a swamp by one's own hair.

At the same time, TCP/IP doesn't see use only in traditional networks consisting of full-size computers, big and important enough for each of them to be managed by a person. Very soon all smart home appliances and personal gadgets may start speaking IPv6 to each other, which is already present as a trend today. Managing such networks will require an unconventional approach because their end users can't be expected to obtain a degree in Information Technology or hire a consultant to operate their fridge or phone. In particular, the manual assignment of an IPv6 address to each "digital pet" will be well beyond the typical end user's capabilities. As for the traditional networks, not all their nodes require a static IPv6 address either: Most workstations and terminals don't need one. For them as well as for personal digital devices, *predictable* automatic address configuration will be more than enough.

> Predictability is essential here in order to facilitate troubleshooting when it is needed despite smart autoconfiguration, for we known it will.

So it is high time for us to focus on the automatic configuration of IPv6 unicast addresses. What steps will it consist of? First of all, the node needs to pick at least one IPv6 address for each of its interfaces. That step can be subdivided further because such an address will consist of a subnet prefix

[13] Except in special IPv6 addresses having a binary prefix of `000` in them.

and an interface ID. It is these two basic pieces that the node needs to work out first, and we will start with the latter: the interface ID.

If the interface to be autoconfigured has a unique link-layer address such as MAC-48 or EUI-64, that will be the best candidate to transform into the interface ID by the well-known procedure from Section 2.7. So, with a suitable link-layer address available, this task will be a piece of cake.

In general, the address transformation procedure specific to a particular link-layer technology is expected to be set down in the accompanying IPv6 Over Foo document, such as [121] for *Ethernet*. Nevertheless, the cases of EUI-64 and MAC-48 are also discussed in the main IPv6 Addressing Architecture document due to their ubiquity and importance [21, Appendix A]. Also provided therein are some recommendations about what to do when no unique link-layer address is readily available.

When no link-layer addresses are built into the interfaces, one option is to have the adaptation layer negotiate a unique interface ID for each peer. This approach is better suited to point-to-point links, and we saw it employed for PPP in Section 4.1.2.

As for the subnet prefix, its selection procedure will depend on the required address scope. If the resulting IPv6 address needs to be link-local, the subnet prefix will simply be known in advance: *FE80::/64*. Thanks to that, the node will need no additional information to autoconfigure its link-local addresses.

Although an address block as large as *FE80::/10* was reserved for link-local addresses back in Section 2.3, only *FE80::/64* can be encountered on interfaces because an IPv6 subnet prefix needs to be 64 bits long, as was decided in Section 2.6. In theory, however, the exact format of a link-local address might depend on the link technology and is supposed to be provided by the corresponding IPv6 Over Foo document, e.g., [121] for *Ethernet*.

On the other hand, if scope larger than the link is required, the subnet prefix can be expected to be chosen by the network admin. To let the node know it, it can be either entered into its configuration by hand or advertised in a centralized manner using a protocol. This protocol can be placed on top of IPv6 because, by our line of reasoning, the node was already able to autoconfigure its link-local addresses and so it is ready to speak IPv6 to its neighbors.

The reader may wonder why this protocol has to be different from DHCP (whose IPv6 reincarnation is known as DHCPv6). The point we are about to prove is that NDP is as capable of doing the job and so no extra protocol will be required to supply a basic configuration to an IPv6 node.

Now let's push the question regarding the subnet prefix advertisement mechanism onto our brain stack until Section 5.4.2 and move on to the next problem immediately concerning autoconfigured address handling. Suppose the node has already built an IPv6 address using the link-layer address of its interface and a well-known or preset subnet prefix, such as the link-local one. Unlike IPX, where the interface ID had to be the same as the MAC-48 address, IPv6 has no such requirement. This makes the situation possible where the automatically chosen IPv6 interface ID is already taken by a neighbor node.

Imagine for example that our local node is about to autoconfigure an IPv6 link-local address on its *Ethernet* interface with the MAC address of *00-01-02-03-04-05*. By the rules from Section 2.7, the corresponding Modified EUI-64 will be this: *02-01-02-FF-FE-03-04-05*. Prepending the link-local subnet prefix to it will yield the complete IPv6 address: *FE80::0201:02FF:FE03:0405*. Granted that the original MAC address was unique, there still is no assurance that the resulting IPv6 address isn't used elsewhere: It can already be assigned to another node on the same link, e.g., through static configuration. Therefore it would be unwise to go ahead and use the autoconfigured address without first verifying that it is really unique within the link.

In a perfect world, the potential address conflict in the above example can result only from a violation of Modified EUI-64: The network admin shouldn't be inventing interface IDs with the **U/L** bit set. In reality, however, stuff happens and even factory MAC addresses might clash some unlucky day because of a manufacturing glitch. The best way to handle all such problems will be to detect and report them before they did any harm to the network.

Such an unverified address will be regarded as **tentative**. Is a new protocol needed to check it for uniqueness? In fact, no it isn't: ND will be capable of doing the job after minimal modifications. A rough plan is already clear: The node will multicast a Neighbor Solicitation regarding its tentative address and wait for replies; should a Neighbor Advertisement come through, that will mean the tentative address is already occupied and must not be assigned via autoconfiguration. The devil is in the details though, and there are going to be enough of those to consider the application of ND to this case as a sub-protocol, to be known as **Duplicate Address Detection**, or **DAD** for short [50, Section 5.4].

As we know, IPv4 relied on a gratuitous ARP Request to achieve a similar goal. Although this mechanism had been around for ages, it wasn't until recently that its procedure was standardized under the name of IPv4 **Address Conflict Detection** (**ACD**) [164].

The first obstacle is this: To be able to speak conventional ND, the node needs an IPv6 address already assigned because its ND packets must have a non-zero Source Address. At the same time, the node can assign itself no address until verified with DAD. To work around this chicken-and-egg problem, DAD must be able to operate on an interface even if it has no address yet. For that, let the node conducting DAD *always* send its NS from the unspecified IPv6 Source Address *::* [50, Section 5.4.2] and with no SLLA option included [139, Section 4.3]. So the unspecified Source Address will mark this Neighbor Solicitation out as a DAD NS among regular NS messages from Section 5.1 sent for the purpose of address resolution or NUD.

But how is the DAD soliciting node going to receive a possible NA in response, given that the NA can't be sent to the unspecified destination? To overcome this obstacle as well, let the NA go instead to the All Nodes link-local multicast address, *FF02::1*, if it is in response to an NS from the unspecified Source Address. To enable the reception of such NAs, if any, the node conducting DAD will need to have joined the All Nodes group prior to sending its NS out [50, Section 5.4.2]

Consequently, the IPv6 multicast group management protocol, MLD, will also need to be able to operate while its speaker has no unicast address yet. Let's keep this in mind until Section 6.4.

Since the DAD NS includes no SLLA option and the DAD NA is sent using genuine multicast, DAD can cope with link-layer address conflicts. Had we attempted to "optimize" DAD and had the DAD NA go to the soliciting node's link-layer unicast address, the switched LAN would have forwarded the NA to just one effectively random owner of that link-layer address and DAD would have probably failed. This consideration is of particular importance because duplicate link-layer addresses in the same subnet can't but result in duplicate IPv6 addresses as long as the interface ID is to be derived from the link-layer address.

On the one hand, the DAD NA is solicited; on the other hand, it can also be received by unrelated nodes because it is multicast to All Nodes of the link. The NA therefore must have its **S**-bit cleared $(S = 0)$ if it is sent in response to an NS from the unspecified Source Address. This is yet another peculiarity of the DAD sub-protocol under the ND umbrella.

The last of the main issues posed by DAD is this: What if multiple nodes try to conduct DAD regarding the same unused address at the same time? None of them will get a DAD NA and so all of them will proceed to promote the tentative address to assigned status, won't they? If so, there is a problem in our draft scheme. The unwelcome outcome can be avoided if the nodes performing DAD are to pay attention to incoming NS messages in addition to NA ones. Should a foreign NS regarding the tentative address come through, the reaction to it will be governed by its Source Address. If it is zero, indicating a DAD NS, then there is another node trying to assign the same address and an address conflict is imminent [50, Section 5.4.3]. On the other hand, if the Source Address of the foreign NS isn't zero, it is a regular Neighbor Discovery message that must be silently ignored because the local node has no rights to the tentative address yet. Of course, to be receiving the relevant NS messages while in DAD, the node will need to have joined the solicited-node multicast group corresponding to the tentative address [50, Section 5.4.2].

The technical fact is, multicast packets sent by the node may be looped back by the link-layer driver, the interface hardware, or the link itself. In the case of DAD, such packets coming back can't be identified by their IPv6 Source Address because it is left unspecified. To avoid a false positive, the node needs some other way to filter its own DAD NS messages out. One option is to use the link-layer source address for the purpose if accessible [50, Appendix A]. Another option, applicable if the node can be certain it is getting back its own multicast, is to count separately the DAD NS messages out and in. Should there be a disparity between the counters, there is an address conflict brewing. In any event, the reliable way to handle this issue has to depend on the knowledge of the particular interface and link's semantics.

An interesting idea suggested by our editor is as follows: The node will be able identify its own NS messages looped back without having to rely on implementation-specific knowledge if a dependable cookie such as an HMAC-based signature is included in them as a special ND option.

As a theoretical optimization, the DAD NA could go back to the solicited-node multicast group instead of the potentially larger All Nodes group. Indeed, the node initiating DAD must join both groups anyway. Although there are no obvious problems with this optimization, it still isn't mentioned in the RFC documents. We would guess that the IPv6 architects worked through this cluster of problems the same way we just did: First they addressed the issue of DAD NA delivery to the soliciting node and only then did they handle the issue of concurrent DAD by multiple nodes. After that, the former solution could have been optimized in the light of the latter one, but it was never done for whatever reason [165].

Now let's bring all these considerations together in a conceptual DAD procedure where node A is to verify a tentative address T on interface I:

1. Node A joins these multicast groups on interface I:
   (a) All Nodes of the link: *FF02::1*;
   (b) the solicited-node multicast group G for address T.
2. Node A starts background reception of ND messages.
3. Node A sends a DAD NS, that is, an NS with an unspecified Source Address and a Target Address of T, to the solicited-node multicast address G.
4. Each node already owning address T,[14] if any, responds to the DAD NS with an NA having $S = 0$ and sent to the All Nodes link-local group.
5. Node A refuses to assign address T to interface I and logs an address conflict alert if, before the DAD timer runs out, any of these ND messages is received:
   (a) a foreign NS about the same Target Address T coming from an unspecified Source Address;
   (b) any NA regarding the Target Address T.
6. The DAD test is negative if the DAD timer[15] expires before any such message comes through.

[14] There can be more than one owner if T is assigned to them in anycast mode.

[15] Parameter *RetransTimer*, default setting *RETRANS_TIMER* (1 second) [139, Section 6.3.2].

Since ND packets can be lost in transit and it is the reception of no relevant ND messages that indicates the tentative address is free and OK to autoconfigure, the DAD procedure should ideally be repeated a few times[16] for the DAD test to be more reliable.

> By default, the DAD procedure isn't repeated because *DupAddrDetectTransmits* defaults to 1, meaning just one attempt [50, Section 5.1].

Equipped with such a nice procedure for DAD, the node can verify any IPv6 unicast addresses to be assigned, not only those auto-created. Thus DAD will provide for early detection of network configuration errors that would otherwise result in duplicate addresses. So it makes perfect sense to demand that *by default any IPv6 unicast address, regardless of its origin, be subject to DAD prior to its assignment to an interface* [50, Section 5.4].

> DAD can still be administratively disabled on an interface via setting its *DupAddrDetectTransmits* tunable parameter to zero [50, Section 5.4].

> Note that DAD is applicable only to unicast addresses. DAD is never invoked on anycast addresses because they aren't unique by definition [50, Section 5.4]. At the same time, DAD is able to detect and prevent the problem case where the local node is about to assign some address A to its interface in unicast mode although the same address A is already assigned to other interfaces of the link in anycast mode. As we discussed in Section 5.3, the knowledge that a particular IPv6 address is anycast is confined to its owner nodes while to the rest the address will be apparently unicast. In the case of a potential conflict the local node will receive NA messages with $O = 0$ and be able to stop. For this mechanism to be reliable, the following condition needs to be met: $MAX\_ANYCAST\_DELAY\_TIME < RetransTimer$.

> If events come about back to front and node X tries to grab address A in anycast mode while neighbor Y already owns it in unicast mode, node X will succeed in the assignment of the address, but all packets for A will still be sent to node Y by its neighbors. Discuss whether automatic detection of this case is possible. (Hint: *Keep in mind there can be ND proxies on the link.*)

[16] Parameter *DupAddrDetectTransmits*, default 1, i.e., just one attempt [50, Section 5.1]

DAD is agnostic to the underlying link technology because it relies on ND, which supports any link with multicast capability in addition to conventional broadcast links such as *Ethernet*. This makes DAD applicable to all links ND can run over, making for a case where ND can be really useful even over a point-to-point link.

Its reliance on multicast can prevent DAD from discovering address conflicts on an NBMA link because the node may not be able to reach all of its neighbors with the DAD NS message. In turn, this can't but create problems for protocols relying on DAD. One such group of protocols we are going to meet will include those concerned with generation of interface identifiers to certain requirements. Needless to say, such protocols simply must invoke DAD on addresses they create. SLAAC from Section 5.4.2, Privacy Extensions from Section 6.2, CGA from Section 6.3, and HBA from Section 6.8 all fall in this group. We will discuss them in due course and for now let's keep in mind that a DAD check is a must for any autoconfigured address.

Perhaps the only good reason to skip DAD is having another protocol work to ensure the addresses are unique. One such example we already know is that of PPP from Section 4.1.2. On a pair of connected PPP interfaces, the autoconfigured link-local addresses will contain unique interface identifiers negotiated with IPV6CP, rendering DAD on those addresses redundant [124, Section 5]. Other IPv6 addresses assigned to the same PPP interfaces may still need to be verified with DAD [124, Section 5].

With DAD fully operational, the IPv6 node can do as much as autoconfigure link-local addresses on its interfaces. Thus the IPv6 node will need zero manual configuration to satisfy the requirement from Section 2.5 that each IPv6 interface have a link-local address [21, Section 2.1]. This is an important step toward the autoconfiguration of IPv6 addresses of larger scope because it enables the node to speak IPv6 with its direct neighbors.

Besides, having link-local addresses will be sufficient for the routerless network from Fig. 2.2 to operate. Effectively, a byproduct of this section's work is an IPv6 mechanism comparable to the dynamic configuration of *169.254.0.0/16* addresses in IPv4 [35].

5.4.2 Prefix and Router Discovery

Now a long-standing item on our to-do list can be fulfilled: to provide for centralized control of the IPv6 host configuration within a link.

As a first step toward this goal, let one or more nodes be appointed responsible for advertising the IPv6 subnet prefixes assigned to this link. By what was said back in Section 5.2, this information should be available to the routers of the link, making them ideal sources of such advertisements and rendering any auxiliary sources such as DHCP servers redundant, at least at the basic level.

> As we mentioned in Section 5.4.1, the reimplementation of DHCP for IPv6 is known under the name of DHCPv6 [166].

Consider a trivial scenario in which just one dedicated router serves the link by advertising the prefix information to the hosts and being selected by the hosts as their default router. Since this scheme operates within the boundaries of a single link, it will be natural to implement its protocol aspect as yet another subset of ND [139, Section 6]. In a nutshell, it can operate as follows:

- The router periodically multicasts a special ND message, to be known as a **Router Advertisement** or just **RA**. Because there is no All-Hosts multicast group in IPv6 and the other routers, if present, may also benefit from hearing such advertisements, the RA will be sent to the All-Nodes link-local multicast group, *FF02::1*.

- A host can prompt a Router Advertisement before the next periodic update is due by sending another special ND message, a **Router Solicitation** (**RS**). Since the host can't be expected to know the IPv6 addresses of all routers present on the link, IPv6 multicast comes handy again: The RS just needs to be sent to the All-Routers link-local multicast group, *FF02::2*.

Keeping our immediate goal in mind, the Router Advertisement should be able to include a list of subnet prefixes available to the hosts of the link for address autoconfiguration. Can we think of other useful information for the RA to carry as well?

It was planned in Section 5.2 that the Default Router List and the Prefix List would also be autoconfigurable in IPv6 hosts. Before the IPv6 autoconfiguration protocol has taken shape, however, we should realize that self-advertising by routers, prefix information advertising, and address autoconfiguration are related yet separate tasks under the ND umbrella. We therefore shouldn't cast them in one rigid brick. So let's consider each task individually before we bring all of them together in the RA message format.

Router Discovery is an interesting feature that existed but never really caught on in IPv4 [167]. In practice, there were two established ways to let an IPv4 host know the address of its default router: It was either set administratively[17] or selected through a routing protocol such as RIP or OSPF. And were we to provide for multiple IPv4 default routers on a link for the sake of redundancy and fault tolerance, the use of a routing protocol would be our only option, entailing the extra overhead of setting up and maintaining a RIP or OSPF speaker on each host.

In contrast with that, the reference IPv6 host has already got a nice mechanism, NUD from Section 5.1, it can use to select a live router from a list using no other protocols but the plain vanilla ND. Thus the IPv6 host is ready to support a list of default routers in place of a single entry, as was discussed in Section 5.2.

> Note that we speak of a list of routers' unicast addresses, not of a single anycast address backed by multiple routers.

The Default Router List comes at a price though: It needs to be configured and maintained, and going from just one router address setting to N of them means an overhead roughly N times as high. Can we turn this job over to the IPv6 host itself? In principle, a Router Advertisement packet coming from a neighbor can be viewed as a sign that the neighbor is a usable router. If it were always the case, the host would just need to add the IPv6 Source Address of each incoming RA to its Default Router List unless already listed. However, to stick by our commitment to make the scheme flexible, a node must be allowed to advertise some useful information with RAs without necessarily having to become a default router for the hosts of the link. The RA format therefore will need a field indicating whether its source wants to be a default router or not.

The next of the RA-based mechanisms penciled in was **prefix discovery**. Are there any uses for it other than that in address autoconfiguration? That is to say, which prefix types can be of interest to the IPv6 host besides subnet prefixes? One such type is on-link prefixes because, as we discussed at length in Section 5.2, in IPv6 they aren't the same as the subnet ones: A subnet prefix isn't automatically on-link while an on-link prefix can be longer or shorter than, or completely different from, any of the subnet prefixes assigned to the link. Back then we envisaged that this information could be distributed by the routers if there were a suitable mechanism for that.

[17] This could be done in a centralized manner with BOOTP or DHCP.

Now we happen to be working on a mechanism that can accommodate on-link prefix advertising easily. For that, each IPv6 prefix listed in a Router Advertisement just needs to be accompanied by some attributes rather than go by itself. In particular, a prefix can be:

(a) suitable for address autoconfiguration;

(b) on-link;

(c) both.

Prefix features a and b essentially independent, either deserves a separate flag bit. Let the autoconfiguration availability bit be denoted by **A** and the on-link bit by **L**.

> The case where both flag bits are unset ($A = 0$ and $L = 0$) would make no sense here, but it still can come handy in a future protocol extension when new prefix attributes need to appear.

Although our model scenario was initially limited to just one router per link, in reality there can be multiple routers advertising prefix information to the hosts of the link. Consider the extended scenario shown in Fig. 5.33, where a link is served by two routers, each responsible for an on-link prefix of its own. So the information about which prefixes are on-link is decentralized, neither router having a complete view. Should, for example, host A send a packet for host B via its default router $R1$, the latter won't be able to send a Redirect back because it has no idea *2001:DB8:777::/64* is on-link, as known to $R2$. Without Router Advertisements, the packets from A to B will have to take the roundabout path via both routers and the Internet even though A and B are neighbors!

Suppose now that Router Advertisements have been implemented and enabled in the same network setup from Fig. 5.33. They are multicast to All Nodes of the link, and so host A will be getting them from router $R2$ as well as from $R1$. Then it will be up to $R2$ to optimize the traffic path from A to B by advertising *2001:DB8:777::/64* as on-link ($L = 1$). Host A will simply add that prefix to its Prefix List, and the traffic it is sending to B will instantly switch over to the shortest path possible. Likewise, host B will get an RA from router $R1$, add *2001:DB8:33::/64* to its Prefix List, and thus be permitted to send the response packets directly to A instead of via its default router $R2$.

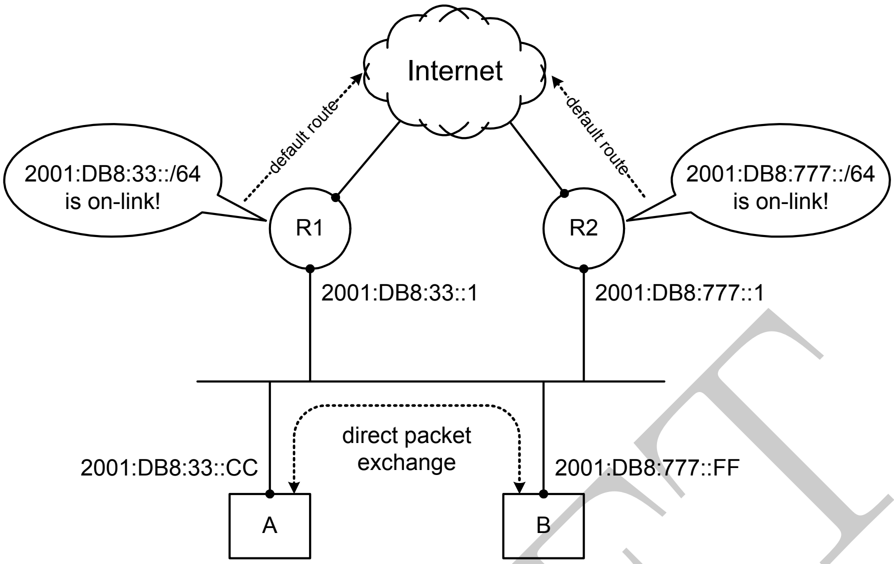


Figure 5.33: Two routers on link, owning different prefixes

Note that a router, unlike a host, must never use RAs from other routers for self-configuration purposes. That said, a router may want to be watching such RAs in order to report any discrepancies between its own configuration and that of its peers.

Great, now the IPv6 host can assign addresses to its interfaces, discover default routers, and learn on-link prefixes in a completely automatic manner—with a little help from its neighbor routers, of course. But what if any of those settings is changed later on? Indeed, prefixes and routers can come and go at the network admin's will. It can't be argued that such changes must be picked up by the network at once so that the admin doesn't have to flush the outdated configuration bits at each host by hand. Without such dynamics, an autoconfiguration mechanism won't be worth a dime.

For the IPv6 autoconfiguration to take on a truly dynamic quality, the host settings need to have a limited lifetime rather than be held on to, once learned, as long as the host is up. The following scheme can do the job [139, Section 6.3.5] [50, Sections 5.5.3 and 5.5.4]:

- Each setting value in the host configuration can have a timer ticking on it.

- Each setting value advertised in an RA message has a lifetime specified therein.

- The host (re)starts the timer on each setting value it (re)learns from the RA received, using its advertised lifetime as the timer interval.
- The host removes a setting value from its configuration if:
  - its timer expires, or
  - an incoming RA advertises the same setting value with a lifetime of zero.

By a setting value we mean an autoconfigured address inheriting its lifetime from the respective prefix with the **A**-bit set, or an entry in the Default Router List, or an on-link prefix in the Prefix List.

Adding a new setting value just learned from an RA is fairly straightforward. A tentative IPv6 address based on a newly learned prefix with $A = 1$ just needs to be tested for uniqueness with DAD before it can be assigned to the interface. A new default router address only needs to be added to the Default Router List so that it can be selected in case the current router is flagged as unreachable by the NUD process. A new on-link prefix ($L = 1$) can be put to use immediately, as soon as it is added to the Prefix List.

> See Section 5.2 to refresh your memory on the Default Router List and the Prefix List. DAD was discussed in Section 5.4.1.

> With each data structure of the reference IPv6 host bound to a specific interface as was shown in Fig. 5.23, it is natural for an incoming Router Advertisement to affect only the context of the interface it was received on.

Will the removal sequence be as simple for all of the settings? The latter two setting types, default routers and on-link prefixes, can only influence the instant decision by the host which next hop to use for an outgoing packet; they don't affect the long-term state of the host. Moreover, the departure of a default router or on-link prefix is an unconditional event the host can do little about: It has to comply right away. These two setting types therefore can be deleted as soon as the timer expires or an incoming RA changes their lifetime to zero.

In contrast to that, a local IP address is a more delicate thing to handle. First, it is bound to by upper-layer protocol sessions, which can exist for an extended period of time, and an abrupt removal of the local address can't but abort all of them in a completely ungraceful manner. Second, it is the host itself that manages its addresses, even if it does so based on external information such as the administrative configuration or Router Advertisements. Owing to that, the host must be allowed to handle the departure of an expired address using a procedure that is only slightly more complex but considerably less disruptive than an instant removal.

For an IPv6 address departure to go smoothly, let the host part with the address in two steps [50, Sections 1 and 5.5.4].

> This is one of the rare cases where computers are unlike humans. For a human, having to kiss something goodbye in stages would be a decidedly more painful experience.

As long as its lifetime hasn't expired yet, the address is available to new upper-protocol sessions[18] as well as usable by the existing ones bound to it. Such an address is said to be **preferred**. Once its lifetime has expired, the address becomes **deprecated**: It can no longer be used to establish new sessions from it, but the already open ones will proceed until normal completion—or at least they get a chance to.

> As an exception, a deprecated address still can be used to establish a new session in case there is no suitable preferred address on the egress interface, e.g., if all addresses of the scope requested have expired [50, Section 5.5.4]. In addition, an application is allowed to request a specific local address via the API even if it is deprecated.

> All this applies only to outgoing sessions. An incoming session to a deprecated address is allowed by default because the remote peer has no way to know the destination address status.

After a while, the deprecated address becomes **invalid** and actually disappears from the interface; any sessions still open from it have to be aborted. To distinguish it from an invalid address, let's refer to an address still assigned to an interface as a **valid** address. Valid are preferred and deprecated addresses.

In this scheme, there are effectively as many as two timers ticking on each assigned address, a **preferred lifetime** timer and a **valid lifetime** timer. Quite naturally, the preferred lifetime of an address can't extend beyond its valid lifetime simply because an invalid address can't remain preferred.

[18] The definition of a session is up to the respective upper-layer protocol.

Tentative state of an address, discussed in Section 5.4.1, can be lined up with its preferred and deprecated states as follows. A tentative address isn't quite assigned to the interface: It can't be used to send or receive packets and the host must not respond to non-DAD Neighbor Solicitations about it. Once DAD is complete, the address is either promoted to preferred state or blocked as duplicate. For the address lifetimes to include the time spent in tentative state, the lifetime timers will have to be started as early as before DAD. Then the implementation will need to allow for the case where either timer expires before DAD is finished.

As an exercise, study the DAD procedure optimization known as Optimistic DAD [168], discuss its pros and cons, and conclude whether you would use it if you were to implement an IPv6 host.

For implementations to be able to handle both static and autoconfigured IPv6 addresses using the same code, let's make one lifetime value special: to mean an infinite lifetime. Since the address lifetime will be encoded as an unsigned integer field in Router Advertisements, it can have a maximum value—represented by "all ones". So let that maximum value mean infinity. Then each local address will have a uniform set of attributes in the host memory, but some addresses will be able to exist indefinitely.

An address of a limited lifetime won't need to cease to exist either as long as there is a router advertising this prefix with the **A**-bit set. As we said above, an incoming RA causes the timers to be restarted on all of the relevant objects. For example, if a local address is due to become deprecated in 5 seconds and an RA is received advertising its subnet prefix with the **A**-bit set and a preferred lifetime of 1 hour, the address will stay preferred for another hour. Thus periodic Router Advertisements can be used to refresh the autoconfigured addresses on all hosts of the link as long as necessary.

The resulting lifecycle of an IPv6 unicast address is shown in Fig. 5.34. In it, invalid state and unassigned state aren't really different; the address can be thought to transition from the former to the latter as soon as its removal is complete: the sessions cleared, the data structures freed, etc.

Now that the requirements to them are known, the formats of RS and RA messages can be composed. The Router Solicitation is nothing but a request for the routers to send their respective Router Advertisements sooner rather than later [139, Section 4.1], its primary information carried by the very fact that the RS has come through. Nonetheless, our experience with the Neighbor Solicitation format from Section 5.1 tells us that it can be a good idea to leave room for an SLLA option in the RS format, too, as

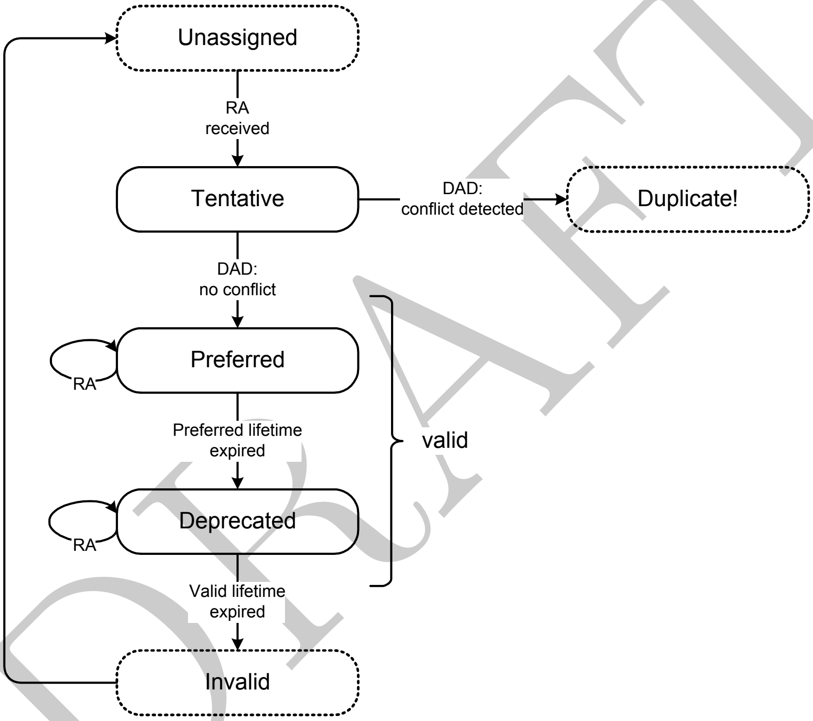


Figure 5.34: IPv6 unicast address lifecycle

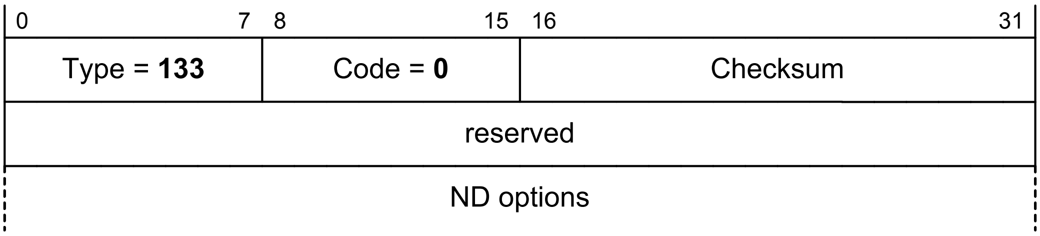


Figure 5.35: Router Solicitation format

shown in Fig. 5.35. Indeed, a host's soliciting router info is often followed by network conversations originating from that host. The routers can try to be ahead of the events by remembering the host's link-layer address in a *STALE* Neighbor Cache entry so that the subsequent packets going back to the host aren't delayed for a round of ND. Besides, an RS with an SLLA option can be instantly responded to with a unicast RA to avoid bothering all IPv6 nodes of the link.

> A router is allowed to unicast its RA in response to an RS from a valid Source Address [139, Section 6.2.6]. Unicasting solicited RAs can, for one thing, extend battery life of IPv6-enabled handhelds due to less multicast "noise" on the link.

> Of course, including an SLLA option makes sense only unless the IPv6 Source Address of the RS is unspecified because only a broken implementation would cache the unspecified address or send a unicast packet to it. By the standard, an RS is permitted to come from an unspecified Source Address, *::*, to work around any chicken-and-egg situations [139, Section 4.1]. Still, nothing seems to prevent a compliant IPv6 host from autoconfiguring a link-local address first and sending an RS from it. Or, to put it differently, should the host have problems with autoconfiguring a link-local address, it will be very likely to run into the same problems when trying to autoconfigure other addresses.

The Router Advertisement has a richer format [139, Section 4.2]. Firstly, we have decided in this section that it needs to have a **Router Lifetime** field in it. A zero lifetime value will indicate that the router doesn't want itself selected as the default router; it is to be removed from the Default Router List if listed, instead of added to the list.

> The base protocol assumes that all routers are equivalent. At the same time, there is an ND extension [169] to assign different priorities to routers as well as to advertise more specific routes to certain prefixes. Hosts supporting this extension may need an additional data structure: the routing table.

Next, the IPv6 Router Discovery mechanism can also be used to set ND and IPv6 host parameters in a centralized manner if the RA format accommodates the relevant fields. We don't really need to discuss here why some parameters have been chosen for autoconfiguration over the others. Instead, we will just present the list of those included:

Current Hop Limit The default value to set **Hop Limit** to when originating an IPv6 packet. (It will make sense to let applications request a different value per socket via the API.) See Section 3.2.

Reachable Time The base value, in milliseconds, used by NUD to compute the random *ReachableTime* (the timeout interval on a *REACHABLE* NC entry). Translates to the *BaseReachableTime* node parameter. See Section 5.1.

Retrans Timer The interval, in milliseconds, used by the basic address resolution with ND as well as by NUD and DAD to time out when waiting for response to a Neighbor Solicitation. Translates to the *RetransTimer* node parameter. See Section 5.1 and Section 5.4.1.

If the router doesn't want the hosts to set a specific parameter from its RAs, it will set its advertised value to zero.

> Reachable Time and Retrans Timer have been chosen probably because we would want to tune those parameters first of all, were we to optimize, e.g., a link with several thousands of hosts on it. On such a link, the NUD overhead can be controlled by increasing Reachable Time while the delay in the response to Neighbor Solicitations caused by their high rate may call for a longer Retrans Timer interval.

> Those parameters could have been implemented as ND options, the same way as the Link MTU parameter was actually handled—see below.

> If a host has more than one interface, those parameter values are associated with the interface the RA came via [139, Section 6.3.2].

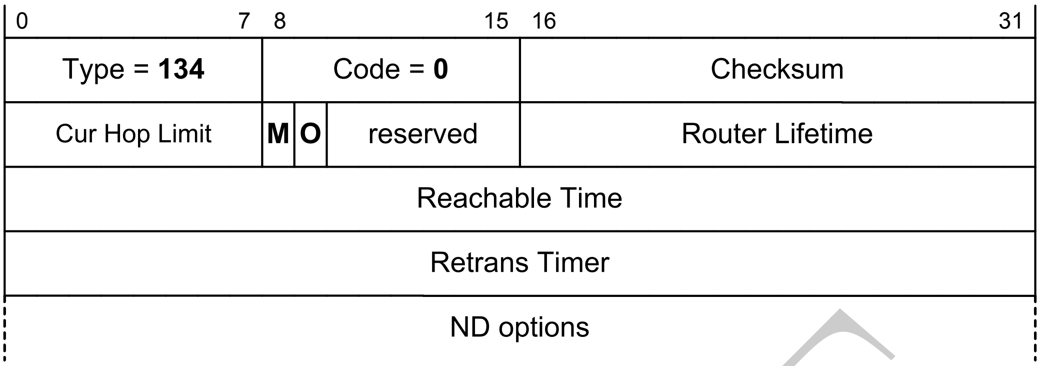


Figure 5.36: Router Advertisement format

> Besides, the RA format includes two flag bits, **M** and **O**, to refer the hosts to DHCPv6 for more settings. The **M**-bit, if set, indicates that an IPv6 address can be obtained through DHCPv6 rather than autoconfigured whereas the **O**-bit set to 1 means that some additional configuration is available through DHCPv6 as well, such as the addresses of DNS and SIP servers, etc. Since DHCPv6 will return all available configuration at once, the **O**-bit is redundant when the **M**-bit is set.

That was the fixed part of the Router Advertisement. Now let's recall that all this work was started to provide for advertising a list of prefixes. Due to its having a variable length, such a list should be implemented using ND options. The overall format of the RA message thus ends up as shown in Fig. 5.36.

To avoid reinventing the wheel, i.e., introducing a substructure within an ND option, let each prefix entry just occupy a separate option. Hence the name of this ND option type: **Prefix Information Option**, abbreviated to **PIO** [139, Section 4.6.2]. If more than one prefix needs to be advertised, multiple PIOs will appear at the tail of the RA.

As planned, each prefix will have the following attributes apart from its 128-bit binary value: the length; the valid lifetime; the preferred lifetime; the **A** and **L** bits, indicating what the prefix can be used for (address autoconfiguration and Prefix List maintenance, respectively). The resulting format of a PIO is shown in Fig. 5.37.

Another ND option meaningful in an RA is SLLA. If included by the router, it will allow the hosts to create a *STALE* NC entry for the router as soon as the RA comes through. This will warm their Neighbor Caches up and eliminate a delay when a packet needs to be sent through the router for the first time. Thus the router will hit many birds with one RA stone.

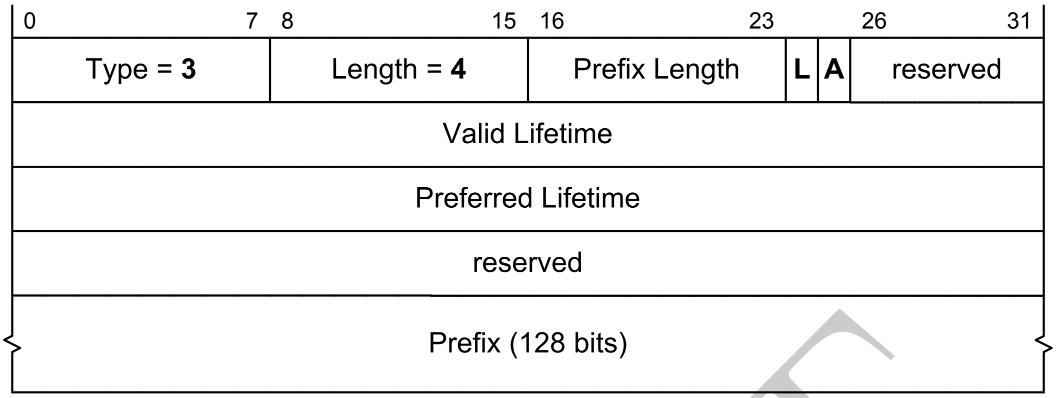


Figure 5.37: Prefix Information Option (PIO)

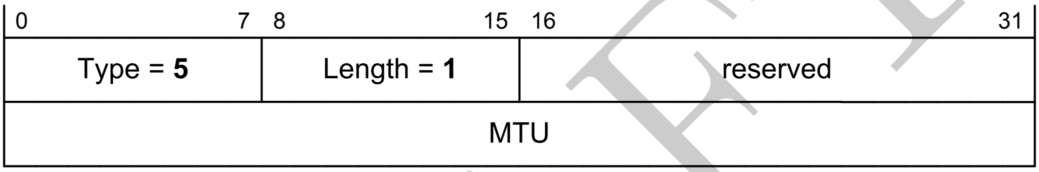


Figure 5.38: MTU Option

The last ND option that can appear in an RA is the MTU Option [139, Section 4.6.4], shown in Fig. 5.38. It can help to avoid MTU-related problems when operating IPv6 over a link whose MTU isn't well-known.

Even an *Ethernet* link can have no single MTU value. Suppose there are two *Ethernet* switches, one with jumbo frames enabled (A) and the other with them disabled or unsupported (B). If connected back to back, these switches will constitute a LAN where the effective MTU between a pair of ports will depend on which switch each port is in. Namely, if both ports are in switch A, the available MTU will be as large as 9000 bytes; otherwise it will be just 1500 bytes. Now suppose that all interfaces plugged into switch A have their MTU set to 9000 while all interfaces connected to switch B have an MTU of 1500. The TCP MSS and PMTUD mechanisms will have trouble coping with an MTU mismatch of this kind because the switches, naturally, will return no Packet Too Big notification and all large frames traveling from A to B will be silently dropped. (The last hope will be the PMTUD black hole detection.) To avoid this problem altogether, a uniform MTU value of 1500 bytes can be advertised to all IPv6 hosts of the link.

Which IPv6 Source Address will a Router Advertisement be sent from?

This question is important because it is this address that will end up in the hosts' Default Router Lists if the **Router Lifetime** is greater than zero. The best choice will be an address least likely to change. This requirement is met by a link-local address because it doesn't depend on the address plan, the current ISP, etc. Therefore a Router Advertisement must be sent from a link-local Source Address [139, Section 4.2].

Of course, this link-local address is hardly arbitrary: It must be owned by the router interface the RA is sent through.

The same rule applied to Redirect messages in Section 5.2 [139, Section 4.5].

It is interesting to observe how the scoped address architecture from Section 2.4 works in the case where RS and RA messages are exchanged. When sending an RS, a host may use any IPv6 unicast address assigned to the sending interface as the Source Address [139, Section 4.1], and a router can choose to respond to the RS with a unicast RA sent to that address [139, Section 6.2.6]. Thus the unicast RA can end up having the Source Address of one scope and the Destination Address of another scope. That is, the Source Address of the RA will always be link-local, but its Destination Address can be, e.g., global. Nevertheless, the RA isn't going to violate the zone isolation principle from Section 2.4 because it will never leave the link.

Similar considerations apply to a Redirect message. Its Destination Address has to be identical to the Source Address of the triggering packet sent through the router, and being such, it is likely to have scope that can span multiple links. At the same time, the Source Address of the Redirect packet must be link-local.

Using Router Advertisements to convey IPv6 and ND parameter values to hosts can entail a consistency problem. Should there be multiple routers on a link, they can be advertising inconsistent parameter values because of a configuration error. In that case Router Advertisements coming from different routers will be overriding the host settings all the time, and a ceaseless variation of the host parameters is likely to introduce instability in the host's behavior. Therefore it is essential that all routers of the link advertise consistent parameter sets. In particular, the basic parameter values from the fixed part of RA as well as the MTU Option values simply must be the same within a link.

As for the prefix lists, different routers should generally be able to advertise different prefixes, but even here some consistency is due. Namely, if a prefix is advertised by more than one router, its suggested lifetimes should be the same across all routers to avoid host address lifetime fluctuations. At the same time, the prefix flags can be allowed to differ because they can be combined by a host using a bitwise OR operation. For instance, if the prefix *2001:DB8:0:123::/64* is advertised with $A = 1$ by one router and with $L = 1$ by another, the hosts are free to add it to their Prefix Lists as well as to autoconfigure addresses based on that prefix.

An interesting issue is how to revoke a former on-link prefix using a Router Advertisement [139, Section 6.3.4]. Just clearing its **L**-bit in the Advertisement won't have the desired effect because $L = 0$ means "dunno" rather than "off-link". What will work as intended is advertising the prefix with $L = 1$ and a zero lifetime. Of the two lifetimes each advertised prefix has, relevant to on-link status is **Valid Lifetime** [139, Section 4.6.2]. If the prefix is also available for address autoconfiguration ($A = 1$), momentarily setting its Valid Lifetime to zero won't result in the existing addresses dropped right away thanks to the "Two Hours" protection rule discussed below and illustrated by Fig. 5.39. But an even safer trick will be to advertise the same prefix in two PIOs, one with $L = 1$ and a zero lifetime and the other with $A = 1$ and a non-zero lifetime.

To help the network admin in keeping the advertised parameter values consistent, each router should watch other routers' advertisements, compare them with its own, and log any serious inconsistencies found so [139, Section 6.2.7].

And what should an IPv6 host do if Router Advertisements coming from different routers don't agree on the length of a certian prefix? Well, you can keep this question until you need to test someone else's understanding of IP fundamentals, for it is a sly trap. In fact, the length of an IP prefix is its integral part, and so the same prefix simply can't have different lengths. Recall that IP prefixes are essentially bit strings and adding or removing even a single bit from a bit string makes a different one even if that bit was zero, numeric interpretation notwithstanding.

At the same time, a prefix can be contained in another one. For example, the binary prefix `01` is present in the prefix `0100` while the IPv6 prefix *2001:DB8::/32* is found in the prefix *2001:DB8::/48*. The longer IP prefix is more specific because it stands for a smaller set of IP addresses. When such prefixes appear on the same link, there is no ambiguity at all because the larger address set, as represented by the shorter prefix, fully contains the smaller set, as represented by the longer prefix. Still, they will remain distinct prefixes, each having its own lifetime and flags. For example, the on-link prefix *2001:DB8::/32* can expire sooner than the on-link prefix *2001:DB8::/48*, and when that happens, the addresses constituting the difference between the two sets will become off-link.

> There could have been a problem if such prefixes of different length had been available for IPv6 address autoconfiguration. However, we are protected from this sort of problem by the policy rule that an IPv6 subnet prefix length always be 64.

Another minor issue to arise when there are multiple routers on a link is that their immediate response to an RS will result in a traffic burst. This can even be used to amplify a DoS attack if the attacker has connected to the link or gained control of a connected node.

> Router Solicitations and Advertisements can't be sent from off-link thanks to GTSM protection, which was discussed in Section 5.1: Their IPv6 **Hop Limit** must be 255 when received.

Indeed, a router is permitted to multicast an RA to All Nodes of the link regardless of what prompted the RA, the timer or an RS [139, Section 6.2.6][50, Section 5.5.1]. This can enable an attacker or just an errant host to cause a multicast storm. To prevent it, each router must rate-limit its RAs.[19]

However, mere rate-limiting won't preclude unwanted synchronization between routers sending RAs. To break it, some randomness needs to be introduced into RA scheduling. If the RA to be sent is solicited, it needs to be delayed for a random time, and if it is unsolicited, the driving timer interval should be made random [139, Sections 6.2.4 and 6.2.6]. The final touch to this scheme will be to skip sending a solicited RA altogether if it got scheduled after an unsolicited one. Thanks to these precautions, RA multicasting won't lead to problems even in the presence of malicious or misbehaving hosts.

[19] Parameter *MIN_DELAY_BETWEEN_RAS*, default 3 seconds [139, Section 6.2.6].

It is a curious fact that various types of periodic events in a network tend to lock into each other, with the resulting synchronization rarely being welcome [146]. To avoid it, randomness has to be introduced into protocol parameters such as delays, e.g., as was done with respect to *ReachableTime* back in Section 5.1. In turn, a high-quality randomness can't be achieved at random: A solid mathematical base is a must [170].

The last technical question regarding Router Discovery is this: Can a router later resign from its role and become a humble host? By the well-known model [139, Section 6.2.1], router function can be selectively toggled on each interface of a node capable of being a router. For stability and security reasons, it is off by default.

Now suppose router function was at some point enabled on a given interface of some node R, so R has been doing all a router can be expected to do, such as forwarding packets off the link, issuing Redirects, sending periodic Router Advertisements. For R to cease being a router in a graceful manner, all information the hosts have learned from, or about, it should be invalidated first.

The most straightforward information channel is R's Router Advertisement. Its information can be voided as easily as by advertising the same configuration parameters with a lifetime of zero, this cancellation message to be repeated a few times just in case. Node R thus will make the hosts expire the prefix-based configuration (autoconfigured addresses and Prefix List entries) as well as have itself removed from the hosts' Default Router Lists.

At the same time, some hosts may still have DC entries resolving to R. Those entries aren't necessarily linked with R's being a default router for the hosts: They could also be created in a host's memory on behalf of R by another router S, which issued a Redirect with a Target Address of R. Consequently, all of such DC entries can't be removed on a Router Advertisement from R, and so the hosts will keep sending packets via R although it is going to ignore any transit traffic from now on. Had node R completely vanished, the stale entries would have been dropped from DC thanks to their connection with the NC entry for R, to be removed through NUD. In the case at issue, however, node R will remain reachable, causing a traffic black hole.

To work around this problem, the node status change needs to be signaled at the most basic level of ND, that is, in a Neighbor Advertisement from R. This is what the **R**-bit was reserved for in Fig. 5.14 (page 205). Let an IPv6 node set this bit to 1 in its NA only as long as it is an active router on the interface the NA is sent through, and to 0 otherwise. Its neighbor hosts thus will have a more reliable means to detect when it ceases to be a router: As soon as the **R**-bit flips from 1 to 0 in its NA messages, all DC entries resolving to the node in question must be invalidated by its neighbors [139, Section 7.2.5].

> To be able to detect such a status change in a neighbor, an IPv6 host will have to keep track of per-neighbor router status in addition to per-neighbor reachability status. For this purpose, a boolean field, *IsRouter*, is included in the reference NC entry [139, Section 5.1].

> There are direct and indirect criteria indicating that a neighbor is a router [139, Appendix D]. While the only direct one is the **R**-bit set to 1 in a Neighbor Advertisement, the indirect criteria include a Router Advertisement coming from the neighbor and a Redirect in which the Target Address is the neighbor's but the Destination Address is different. At the same time, a Redirect should come *from* the neighbor only if it is already used as a router by the local host. Unlike the direct one, the indirect criteria can't signal the opposite, i.e., the neighbor's being a mere host.

The Router Discovery mechanism complete, we can finally compile a detailed procedure of how an IPv6 host will autoconfigure its addresses. To refresh our memory of what has been said so far, let's start from the moment when the host has just loaded its IPv6 module and enabled its network interface. Then the host proceeds as follows:

1. An N-bit interface ID is created based on the link-layer address of the interface.

   > Let's keep in mind that $N = 64$ is a policy decision, not to be hardcoded in the IPv6 core procedures and formats.

   > Strictly speaking, an IPv6 interface ID doesn't need to be always based on the link-layer address of the interface [21, Appendix A]. However, this is a de facto standard practice for interfaces having an accessible and unique link-layer address.

2. A tentative link-local address is created by concatenating the prefix *FE80::/*$(128 - N)$ and the interface ID created in step 1.

3. That the tentative link-local address from step 2 is unique is verified with DAD.

4. If the tentative link-local address is unique, it is assigned to the interface with its lifetimes set to infinity. Otherwise the procedure has to stop because the host must have at least one IPv6 link-local address on its interface; besides that, IPv6 operation should be disabled on this interface altogether since a link-layer address conflict is likely [50, Section 5.4.5].

5. Possibly, a Router Solicitation is multicast to All Routers of the link, *FF02::2*.

6. As Router Advertisements start coming in, the Prefix Information Options in them are scanned through and handled as follows [50, Section 5.5.3]:

   (a) An option is ignored for address autoconfiguration purposes if it has the **A**-bit unset.

   > An option with $A = 0$ can still be used to populate the Prefix List if it has $L = 1$.

   (b) An option is ignored if the prefix advertised in it is link-local.

   (c) An option is ignored if the Valid Lifetime value is less than the Preferred Lifetime value.

   (d) If the advertised prefix is newly seen:

      i. It is ignored if its length plus the interface ID length N isn't 128, i.e., if they don't make an IPv6 address.
      ii. It is ignored if its Valid Lifetime value is zero.
      iii. Otherwise a tentative address is created by concatenating the advertised prefix and the interface ID from step 1.
      iv. The tentative address is subject to DAD.
      v. If it is unique, the tentative address is assigned to the interface with its lifetimes set as per the PIO received. Otherwise it is marked as duplicate, to wait for some sort of manual intervention.

   (e) Otherwise, i.e., if the prefix has already been used here to create and assign an IPv6 address:

      i. The preferred lifetime of that address is set as per the PIO.

ii. The valid lifetime of that address is set based on the PIO value, subject to a restriction (see the following remark).

> Although we haven't discussed ND security yet, it is already clear that, unless extra protection is in place, ND messages are vulnerable to spoofing. In particular, RA spoofing can be used to mount a trivial DoS attack, in which the victim host is sent a fake RA with very short prefix lifetimes, thus making the host shed its addresses and go unreachable until the next legitimate RA comes through. Note that no flooding of the victim host is need, unlike in a typical DoS attack: A single malicious RA will be enough.
>
> To block this attack vector, an RA shouldn't be allowed to reduce the valid lifetime of an address to too small a value [50, Section 5.5.3]. The technical essence of the restriction is pretty simple. First of all, an RA can extend the lifetime as much, or as little, as needed. E.g., if the lifetime currently left is 5 minutes and the advertised lifetime is only 10 minutes, the new lifetime will be set to the latter regardless. But if the RA is actually trying to make the address expire sooner, the advertised lifetime will be cut off at 2 hours. For example: If the remaining lifetime is 1 hour and the RA tries to change it to 10 minutes, the address will live for another full hour. If the remaining lifetime is 3 hours and the advertised lifetime is 45 minutes, the new lifetime will be set to 2 hours. And only if the advertised lifetime is over 2 hours, it will have verbatim effect. This is graphically illustrated in Fig. 5.39.
>
> When a stronger ND security mechanism is made available in Section 6.3, this restriction will become optional. That is to say, it will need to be applied only to unauthenticated RA messages. In addition, it applies only to the valid lifetime because it is a more vulnerable parameter, whereas the preferred lifetime can be manipulated freely by RAs.

The host will need to keep executing step 6 of the above procedure in the background as long as its interface is running because periodic RAs tell the current configuration of the link, which can be adjusted on the fly by the network admin. A part of the procedure can be of use to IPv6 routers, too; an IPv6 router can execute steps 1 through 4 on each of its interfaces to autoconfigure its link-local addresses; steps from 5 on are for hosts only [50, Section 4].

An interesting feature of this autoconfiguration mechanism is that neither the host nor the router needs to keep persistent information about the addresses configured so. Each time the host is powered up, it runs the procedure from scratch and, as it is easy to see, gets the same address configuration as

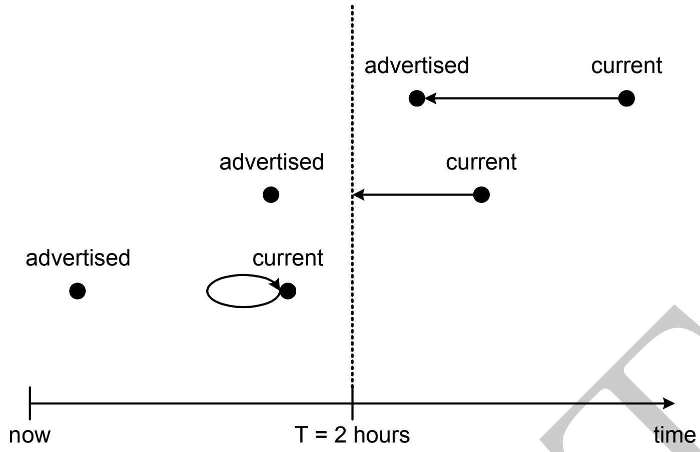


Figure 5.39: Preventing a lifetime-based DoS attack

long as its link-local address and the configuration of the router remain unchanged. In a nutshell, this mechanism features statelessness. Hence its well-known name: **IPv6 StateLess Address AutoConfiguration**, or **SLAAC** for short [50].

> As a minor optimization, an IPv6 host may use non-volatile storage to hold on to its current SLAAC-based configuration values, but only within their respective lifetimes [50, Section 5.7].

An obvious condition for SLAAC to be able to operate over a link is that the link support Layer 3 multicast. Otherwise SLAAC will fail to run over such a link.

> Note that only Layer 3 multicast is needed for SLAAC, whereas real Layer 2 multicast, such as offered by *Ethernet*, isn't required as long as the link layer can emulate it. For example, on a point-to-point link multicasting is as simple as sending packets to the remote station. And if a link is multi-access and broadcast but lacking multicast, IP multicast packets can be sent in broadcast frames, relying on the IP layer to filter them out on input. An example of this approach can be found in *ARCNET* [171, Section 4.3][172, Section 7].

A fairly complex automatic mechanism, SLAAC should be controllable. The bare minimum of such controllability will be the option to disable SLAAC on an individual interface, should it not be desired there for this or that reason. For an IPv6 host this means that it must also be able to work with manual settings, ignoring any incoming Router Advertisements. As for an IPv6 router, its RA sending function needs to be off by default[20] because it can affect many hosts at once; it should be enabled only when actually needed and not sooner than all parameters to be advertised are properly set up.

> The above statement isn't at odds with the mandatory status of SLAAC support in IPv6 nodes. The point is, there is a significant difference between having a mechanism turned off but ready for use and not having it at all. The same applies to DAD [50, Section 5.4] and ND in general: The option to disable a mechanism doesn't make the mechanism itself optional.

> One useful application of SLAAC beyond everyday autoconfiguration of IPv6 nodes is site renumbering [50, Section 4.1], e.g., when changing ISPs. With the IPv6 subnet prefix length fixed at 64 bits for practical purposes, one can keep the interface IDs and change only the subnet prefixes. In each subnet, a smooth transition can be ensured through advertising both prefixes, the old one and the new one, as available for autoconfiguration ($A = 1$) during the transition period. So each host will assign its interface two addresses, an old one and a new one. Next, the old addresses can be phased out by making them deprecated through the Preferred Lifetime attribute of the old prefix. Once deprecated, they will no more be used to establish new upper-layer protocol sessions such as TCP connections. Finally, the old addresses can be completely removed through manipulating the Valid Lifetime of the old prefix. An advantage of this technique is that only the parameter values advertised by the routers need to be adjusted while the hosts will pick the changes up from Router Advertisements. More details of the seamless renumbering procedure can be found in [173].

[20] Parameter *AdvSendAdvertisements*, default *FALSE* [139, Section 6.2.1].

Chapter 6

Advanced IPv6

6.1 Anycasting to the Routers of the Subnet

Back in Section 5.4.2 a mechanism was "built" for an IPv6 host to populate the list of default routers serving this link. Using that mechanism the host would learn the IPv6 unicast addresses of such routers. Afterwards it could apply this or that selection policy to the resulting Default Router List, preferring one of the routers to the others.

On the other hand, should the host not really care which router gets selected as long as it is reachable, anycast from Section 5.3 becomes applicable. To be specific, the same anycast address needs to be assigned to all suitable router interfaces into the link and made known to the hosts concerned. The anycast address being indistinguishable from a unicast one, it can just come from the link subnet. If its interface ID is nailed down at a well-known value rather than chosen per link by the network admin, the hosts will be able to work out the **Subnet Router Anycast Address** from the subnet prefix and no extra host setting will be needed. Let this special interface ID have a numeric value of zero [21, Section 2.6.1], as shown in Fig. 6.1. Consequently, *the IPv6 interface ID of all zero bits is reserved and can't be used in regular IPv6 unicast or anycast addresses.*

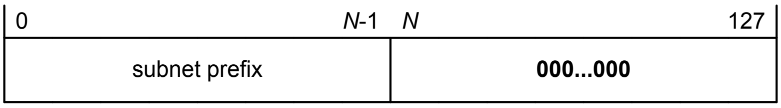


Figure 6.1: Subnet Router Anycast Address format

Originally [21, Section 2.6], the Subnet Router Anycast Address was singled out not only to provide for host-to-router communication within a link, but also to let an external host construct a packet so that it would arrive at the link boundary, e.g., thanks to a Routing Header from Section 3.3.3. Besides a link, this plan also tried to accommodate larger administrative entities such as a routing domain. To the best of our knowledge, it has seen no practical use.

A rather special case where Subnet Router Anycast Addresses may actually be used by some network operators is the troubleshooting of tunnels; in particular, that of 6rd [174, Section 5]. In a nutshell, the tunnel endpoints thus can access each other, e.g., *ping* the remote peer, using the same predefined address looking like a unicast one, the latter quality coming handy when the troubleshooting tool doesn't support multicast.

Additional well-known anycast interface IDs were introduced later. A general rule is that the highest 128 interface ID values available in a subnet are reserved for this purpose [161]. In subnets where interface IDs need to comply with the Modified EUI-64 format from Section 2.7, this means that the **U/L** bit has to remain unset in the well-known anycast interface IDs as they aren't globally administered by *IEEE*. The Subnet Router Anycast Address predated this general rule and so it is an exception. It is interesting to note that the **I/G** bit in those interface IDs except the zero ID is set to 1, which can be thought of as a reminder that an IPv6 address based on such an ID won't quite be a unicast one.

That the Subnet Router Anycast Address has already been standardized is a valid argument against using */127* subnet prefixes on point-to-point links [175, Section 3]. If such a subnet prefix is assigned to a point-to-point link, one interface inevitably gets an ID of zero. To make things worse, the other interface can have the genuine Subnet Router Anycast Address assigned to it, differing only in the anycast flag local to the node. Thus an address conflict will ensue because an anycast address is never subject to DAD from Section 5.4.1. The counterargument is that Subnet Router Anycast Addresses see very limited use [53, Section 4]. (Taking neither side here, we suggest the reader make a decision on a per-case basis, our hope being that by the end of the course he/she will be prepared for that.)

6.2 Privacy Extensions for SLAAC

> BIG BROTHER IS WATCHING YOU
>
> G. Orwell. Nineteen Eighty-Four

> Just because you're paranoid doesn't mean they aren't after you.
>
> J. Heller

The SLAAC mechanism from Section 5.4 can not only make the life of IPv6 users easier but also spur their unease about being followed and watched—and their concern won't be totally unfounded. Suppose a user has got a portable computer (such as a laptop, tablet, or smartphone) with a network interface based on an *Ethernet* or *WiFi* chip and so having a hardwired MAC-48 address. While traveling around the globe, the user would connect to a local IPv6 network at each stopoff, his computer conveniently getting a suitable IPv6 address for its interface through SLAAC. It isn't hard to see that the interface ID in all such addresses will be just the same because it is based on a constant MAC-48 address. For example, if the MAC address is *00-01-02-03-04-05*, the interface ID made from it will be as follows: *02-01-02-FF-FE-03-04-05*.

If you have difficulty seeing why, please review Section 2.7.

Thanks to joint effort by *IEEE* and *IETF*, this ID will be globally unique, identifying the user's mobile computer. So the user will effectively have a digital ID tag planted on him as long as he carries his "silicon pet" around the world. At the same time, the current IP subnet prefix can often tell the user's whereabouts due to many IP address assignments bearing geographical locality. The user thus can be easily tracked by his current IPv6 address, which is no secret to any Internet resource the user hangs out at—by the very design of TCP/IP. For example, the user's itinerary can be reconstructed with a minimal effort from the log file of any web site frequented by him. Finally, linking the digital ID to the personal identity is often made possible by the user himself, e.g., through entering real personal data in a web forum registration. An uncomfortable conclusion is that today we can be watched not only by three-letter agencies but also by curious individuals with basic sysadmin rights to an Internet server.

Needless to say, everyone should decide themselves if this threat is real for them, while our task is just to provide a mechanism to thwart the threat when necessary. The good news is that almost all tools we are going to need for the job are already at hand.

Let's start off with pinpointing the culprit: a constant and globally unique interface ID. As a matter of fact, neither of its qualities is required by IPv6: We just got from *IEEE* far more than we asked for when mere link-scope uniqueness was needed for the interface ID in Section 5.4. On the other hand, even those qualities were insufficient to ensure that an autoconfigured IPv6 address was born unique; the DAD procedure from Section 5.4.1 was still required. So can we now rely solely on DAD instead of the guarantees by *IEEE*?

Suppose an IPv6 host were to generate a random, unpredictable interface ID each time it needed to perform SLAAC on its network interface. Such an ID may already be taken by a neighbor, but this problem can easily be detected with DAD. Now, what if the DAD test came out positive, meaning a duplicate address? In the regular SLAAC from Section 5.4 this would be a showstopper to resolve by hand because the interface had just one predefined hardware address to make its ID from. Fortunately, now the situation is different: The host can just generate another random interface ID, verify it with DAD, and so on until it has a unique ID. This simple and elegant idea is the essence of what is known as **Privacy Extensions for SLAAC** [176].

> Of course, even a random interface ID needs to conform with the Modified EUI-64 format from Section 2.7. In particular, its **U/L** bit must be unset because such an ID wasn't made from a globally unique EUI-64 or equivalent [176, Section 3.2.1]. At the same time, the Privacy Extensions RFC document doesn't require the **I/G** bit be cleared and we don't think we know a reason for that; it might be just an oversight. For one thing, the CGA format from Section 6.3 also concerned with creating effectively random interface IDs does have this requirement.

> Another two checks for the random interface ID to pass are that it is neither a reserved value nor one already in use by the host. The point of the latter check is that the random interface ID must have no connection with the host's current identity.

Although the current Privacy Extensions protocol assumes 64-bit interface IDs [176, Section 1], its architecture doesn't really depend on a fixed interface ID length since generating a random value of any suitable length is not a problem.

The first tool the host is going to need here is a quality (pseudo)random number generator. If pseudorandom numbers are used, an additional challenge is to prevent two hosts on the link from generating the same sequence. That boils down to seeding the generator with a sufficiently unique and hard-to-predict value. These problems are neither new nor exclusive to Privacy Extensions and there are reliable solutions available for them [170]—no need to reinvent the wheel here.

This is where a globally unique MAC or EUI address can come handy. If hashed along with a local, unpredictable value such as the output from a random noise generator chip, it will contribute uniqueness of global scope to the pseudorandom generator state. (A typical random noise generator chip alone can't suit most real-world applications because of its very limited data rate.)

The Privacy Extensions RFC insists on using a specific hash function-based algorithm to generate a random interface ID [176, Section 3.2], but it must be doing so only to prevent implementations from using bad ad-hoc algorithms. The algorithm choice shouldn't really matter as long as the interface ID generated is sufficiently random.

This mechanism, as we have just roughed it out, is already able to protect the privacy of traveling users who change their point of connection often because a new interface ID is created at each reconnection. Can resident users also benefit from Privacy Extensions? They will be able to providing their desktop hosts are allowed to pick a new random interface ID periodically even if the interface remains connected. At the same time, the subnet prefix (or prefixes) available for address autoconfiguration will be dictated by the network; it can't be changed as freely as the interface ID. Together the given subnet prefix and a random interface ID will form a so-called **temporary** IPv6 address whose overall lifetime is artificially capped to maintain Internet access privacy, as opposed to a regular autoconfigured address that can last indefinitely thanks to refreshing by incoming Router Advertisements.

Let's now extend the IPv6 address lifecycle from Fig. 5.34 in page 263 to incorporate temporary addresses as used by Privacy Extensions. The result will be shown in Fig. 6.2. A temporary address is born in Tentative state because it is subject to a mandatory DAD check. If it proves unique, it will get promoted to Preferred state and become suitable for outgoing session origination. Otherwise a new random interface ID just needs to be generated.

> Just in case, the total number of such attempts to generate a unique interface ID is limited to $TEMP\_IDGEN\_RETRIES$, which is 3 times by default [176, Section 3.3]. Indeed, if multiple attempts fail one after another, there should be an unforeseen fault for the operator to deal with.

A temporary address is bound to become Deprecated sooner or later; it can't remain Preferred beyond what is specified in the host parameters.[1] Once Deprecated, it should no longer be used to establish new outgoing sessions. Therefore it will be a good idea to have generated and assigned a new temporary address in a bit of advance[2] so that the interface always has at least one Preferred temporary address. Since an IPv6 interface is perfectly capable of having multiple addresses, it won't be a problem if the lifetimes of the old and new addresses overlap slightly.

If maximum privacy is required and can be afforded, each new temporary address will need a freshly generated random interface ID. However, if the privacy requirements are more relaxed or the host's computing power is limited, it can be sufficient to generate a random interface ID for each of the host's interfaces on a periodic basis [176, Section 3.5]. The generation period still should be short enough for a new address to be different from the old one using the same subnet prefix.[3]

> Ideally, another event type prompting immediate address regeneration should be the host's reconnecting to a different link [176, Section 3.5]. However, to reliably detect that the link has changed isn't as trivial a task as it may seem [177].

Meanwhile, the old temporary address will hang on in Deprecated state to let the sessions opened from it terminate normally. It has already been

[1] Its lifetime is calculated as $TEMP\_PREFERRED\_LIFETIME - DESYNC\_FACTOR$, where the default $TEMP\_PREFERRED\_LIFETIME$ is 1 day and $DESYNC\_FACTOR$ is a random adjustment made to fight unwanted synchronization [176, Sections 3.3 and 5].

[2] Parameter $REGEN\_ADVANCE$, default setting 5 seconds [176, Sections 3.4 and 5].

[3] That is, it should be less than $TEMP\_PREFERRED\_LIFETIME - REGEN\_ADVANCE - DESYNC\_FACTOR$.

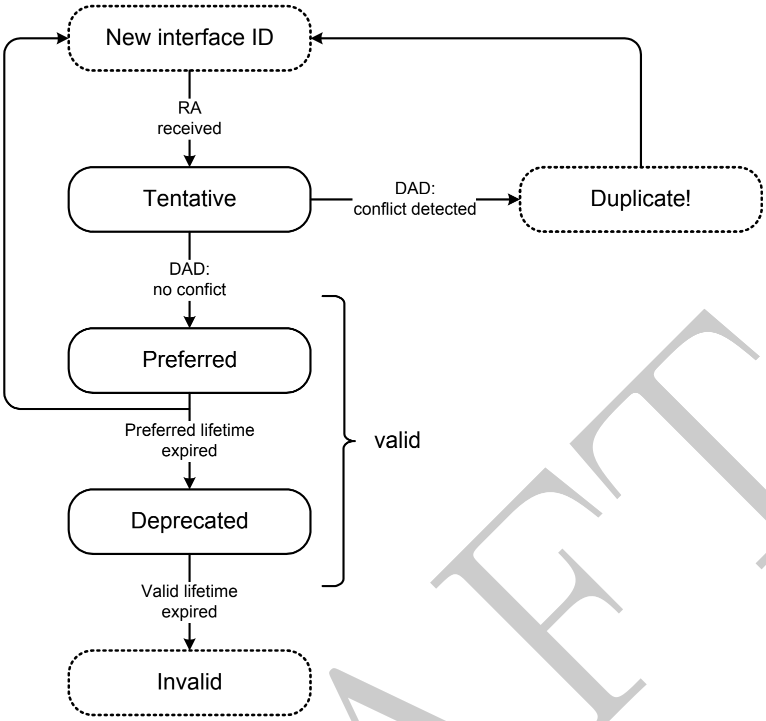


Figure 6.2: The lifecycle of an IPv6 temporary address

exposed to the 'Net, so there would be little sense in forcing its removal before the sessions can be expected to close naturally. At the same time, the old address shouldn't be allowed to linger on forever, e.g., because of a stuck session, because overall that could give rise to buildup of zombie addresses, each of them wasting its share of host and network resources including data structures and solicited-node multicast group membership. Consequently, there has to be a hard limit on the valid lifetime of a temporary address as well, though the associated interval value can be quite liberal.[4]

In other respects a temporary address won't be too different from regular IPv6 addresses. In particular, Router Advertisements can also control its lifetime. This can be pictured as follows. A regular, non-temporary address has just two timers ticking on it, one counting down its preferred lifetime and the other, its valid lifetime; both timers are reset at once if an incoming RA lists the corresponding subnet prefix with the **A**-bit set. At the same time, a temporary address has as many as two *pairs* of timers on it, one pair for preferred lifetime and the other for valid lifetime. In each pair, one timer

[4] Parameter $TEMP\_VALID\_LIFETIME$, default setting 1 week [176, Sections 3.3 and 5].

is still controlled by RAs while the other can be primed only once, when the address is assigned, so its countdown is strictly monotonic. Nevertheless, either timer in a pair can fire first, thus prompting the transition of the address to its next lifecycle stage according to Fig. 6.2.

To conclude this section, we can't but point out how just a slight modification to the basic mechanisms of the IPv6 host was sufficient to provide for address-level privacy, once again demonstrating the power of the IPv6 architecture. Did it buy us much privacy though? Well, we think it bought us enough of it. Granted, the identity of an IPv6 user can be leaked out not only in the address; but when it comes to protecting security or privacy, all layers are important because any of them can be targeted by an attacker. This is why Privacy Extensions, while hardly sufficient alone, are still essential for blocking attacks on privacy in the IPv6 Internet.

That being said, privacy isn't always a concern while Privacy Extensions can make it non-trivial to troubleshoot the network. For this reason, a Privacy Extensions implementation must have an on/off knob defaulting to the off position [176, Section 3.6].

There is an important consideration for network security arising when temporary addresses are in use. If the hosts accept incoming sessions on temporary addresses, extra care will need to be taken when working out security policy rules to protect sensitive applications. In particular, not all host addresses will be known beforehand and so it will make little sense, say, to block access to a single IPv6 address or a fixed list of those.

Another consideration is that some public online resources can restrict access from IP addresses having no *PTR* record in DNS. This problem can be worked around by using DDNS to publish a random name for each temporary address. (Publishing the real host name would ruin privacy.)

6.3 Securing Neighbor Discovery

The ND protocol from Chapter 5 is decidedly more powerful and flexible than its IPv4 counterpart, ARP; still, it remains as susceptible to premeditated attacks. We aren't going to discuss all possible attack vectors here; it will suffice to consider the low-hanging one, where a malicious node appears on the link and proceeds to respond to Neighbor Solicitations from other nodes with fake Neighbor Advertisements mapping the IPv6 Target Addresses solicited to the malicious node's link-layer address. Thus the malicious node can intercept packets for some or all next hops present on the link. For example, if the attacker assumes the default router address, all traffic from the link to the Internet can be intercepted. Since Neighbor Advertisements from the legitimate owner of the stolen IPv6 address can be interfering with the attack, they just need to be suppressed with a DoS attack on that node run in parallel; alternatively, the attacker can make use of a moment when the legitimate node is down, e.g., for maintenance. This attack is based on the same idea as ARP spoofing, so it can be called "ND spoofing".

This and other attack vectors targeting ND are discussed at length in [178].

Using the NUD state machine from Fig. 5.5 in page 193 and Fig. 5.9 in page 199, verify that it will be sufficient for an attacker to send a Neighbor Advertisement with $S = 1$ and $O = 1$ to poison an existing NC entry with a rogue link-layer address although a new NC entry can't be created this way.

Suggest a way for an attacker to force creation of a rogue NC entry. (Hint: *The following ND messages can be used: NS, RS, or RA with SLLAO; Redirect with TLLAO.*)

Unlike ARP, which stood beside IP in the protocol stack, ND is a subset of ICMPv6 and so, in theory, the universal IPv6 security framework as provided by IPsec from Section 3.3.5 can be applied to ND. Unfortunately, practical considerations make this solution less than attractive notwithstanding its high reliability. At the bottom level, the IPsec machinery is controlled by security associations, which are just elementary rules telling how to handle packets from source such-and-such to destination so-and-so. When there are N nodes on a link, each having M addresses, up to $2 \times (N \times M + 1)$ security associations will need to be set up on *each* node before ND can freely operate under complete protection [179, Section 6]. This would be a tedious and unrewarding job of complexity $O(N^2)$. At a higher level, individual security associations can be conveniently managed by the IKE protocol rather than by hand, but IKE requires the basic IP layer be operational, thus resulting in a chicken-and-egg situation. In particular, IKE can't help in securing the most sensitive function of ND, namely, unicast address resolution [180].

Can ND be made secure while keeping manual configuration complexity at $O(N)$? Let's see. The first candidate to consider will be symmetric-key cryptography. Suppose each authorized node kept the shared key in its memory, using it to sign outgoing ND messages and to verify incoming ones. Although this scheme is $O(N)$ configuration-wise, there is a critical weakness in it: As soon as the shared key is compromised, the attacker will be able to impersonate any node. In addition, to re-secure the link after the incident, a new shared key will have to be entered on each legitimate node. All in all this scheme isn't going to cut it.

The other candidate is public-key cryptography, where each authorized node gets a unique pair of keys. Outbound ND messages are to be signed with the node's own private key while inbound ones are to be verified using the relevant neighbor's public key. If a node's private key gets stolen, only that node's identity will be compromised, which is a big advantage over the previous scheme. Still, how can a node learn the public keys of its neighbors? Were they to be distributed by hand, the link configuration complexity would climb to $O(N^2)$. The alternative is for the source of an ND message to attach its public key to the message. However, until the recipient of the message has out-of-band information about its source, the message may as well come from an attacker who just went ahead and created a valid pair of keys. Something is still missing from this scheme.

> We didn't even try to apply a Public Key Infrastructure (PKI) at this point because it would have required IP connectivity. Why so, we will see later in this section, when discussing router authorization.

To be precise, we just missed something when accepting the assumption that no out-of-band information was available about the neighbor that had sent us the ND message. In fact, when our local node invokes ND to resolve a neighbor's IPv6 address, that address has been known since before, and independently of, the ND message exchange. This is also an assumption, but it is one we can build on: that the neighbor's IPv6 address can be trusted. While working at ND level, we can't but need certain areas of trust provided by other layers and protocols.

What is the meaning of an IPv6 address being trusted? It means that the address was learned from a trusted source. Suppose for example that we want to *ping* an interface of neighbor B. How can we find out its IPv6 address reliably? First, we can just walk to B's console, log into the system, and read the address we are after off the screen.

> No doubt, the console can be hijacked as well, e.g., using a rootkit. However, we have to assume it is secure because it is yet another area of trust for us now. Simply put, we can't be solving all security problems at once.

Second, if we can trust DNS[5] and the DNS name of node *B* is known, *ping6* will resolve the name to an IPv6 address for us:

```
$ ping6 b.example.org
PING6(56=40+8+8 bytes) 2001:db8::2 --> 2001:db8::1
...
```

Third, *B*'s address can be told to us by a person we trust, e.g., in a signed email. . . The reader certainly can suggest even more options here.

In any case, *the assumption will be that the target address to be subject to ND is known in advance from a trusted source.* Then nothing prevents us from turning that address into a container for the information still missing. If its subnet prefix can't be arbitrary because it depends on the network-wide configuration, its interface ID is still available for the purpose. We are already skilled in putting strange things there, such as a slightly modified link-layer address in Section 5.4.1 and a random value in Section 6.2; and now it is going to be a fingerprint of the target node's public key. This will enable other nodes to verify authenticity of the key used.

Once this scheme is adopted, the interface ID can no longer be arbitrary because it has to depend on the public key of the node to own the address. The address assignment and publication procedure will be roughly as follows:

1. Generate a key pair for the node.
2. Compute the interface ID as the public key fingerprint using a well-known algorithm.
3. Make an IPv6 address from a suitable subnet prefix and the interface ID just computed.
4. Enter the keys into the node settings.
5. Assign the address to the node interface.
6. Publish the address as appropriate, e.g.:
   (a) Register it in DNS.
   (b) Enter it directly in the configuration of other nodes where required by the application, e.g., if it is going to be a DNS server or a default router.
   (c) Let the interested parties know it.

[5] E.g., thanks to the use of DNSSEC.

This recipe will produce a so-called **Cryptographically Generated Address**, or **CGA** for short. An exhaustive codex of rules governing the use of CGA, to be found in [181], is considerably more elaborate than the sketch we have just come up with. Still, the basic idea behind it is clear and elegant. CGA also has two interesting practical features to highlight.

First, if the node is a user workstation accepting no incoming sessions, its IPv6 address is free to change periodically. For a non-CGA address, this was handled by Privacy Extensions from Section 6.2. Note that step 6 of the above procedure doesn't really apply to such a node simply because its current address is of interest to nobody else.

> Still, it can make sense to register the temporary CGA in DDNS for the reasons discussed in Section 6.2.

The client-only node can perform steps 2, 3, and 5 automatically. Assuming its key pair remains the same and so executing step 1 isn't required each time, such a node will be able to recompute its CGA periodically; but how can the fingerprint be varied if the algorithm and its input are fixed? For that, it just needs to depend on an additional argument known to all peers. This argument, known as the CGA **modifier**, isn't secret and can be transmitted along with the public key. Together with a few more attributes, it constitutes the CGA **parameters**. The fingerprint value depends on all of the parameters, and so they have to accompany the public key when in an ND message.

> One such parameter is the subnet prefix of the CGA. The point is that making the fingerprint depend on the subnet prefix prevents an attacker from taking an authentic CGA and replaying its interface ID on a different subnet.
>
> Another CGA parameter is a collision count. Because a CGA depends on a random modifier, it also comes out essentially random, and so a conflict with an existing IPv6 address can't completely be ruled out. Such a conflict will immediately be detected by DAD from Section 5.4.1, but how can it be resolved without manual intervention? If it happens, the collision count will just need to be incremented by one, followed by the fingerprinting hash function recomputed. Provided the hash function is strong enough, the new interface ID will be completely different. Why just generating a new modifier value isn't an option, we will see when discussing the next CGA feature of interest.
>
> The actual CGA Parameters format is shown in Fig. 6.3. The public key is just included in it to make sure it never comes alone.

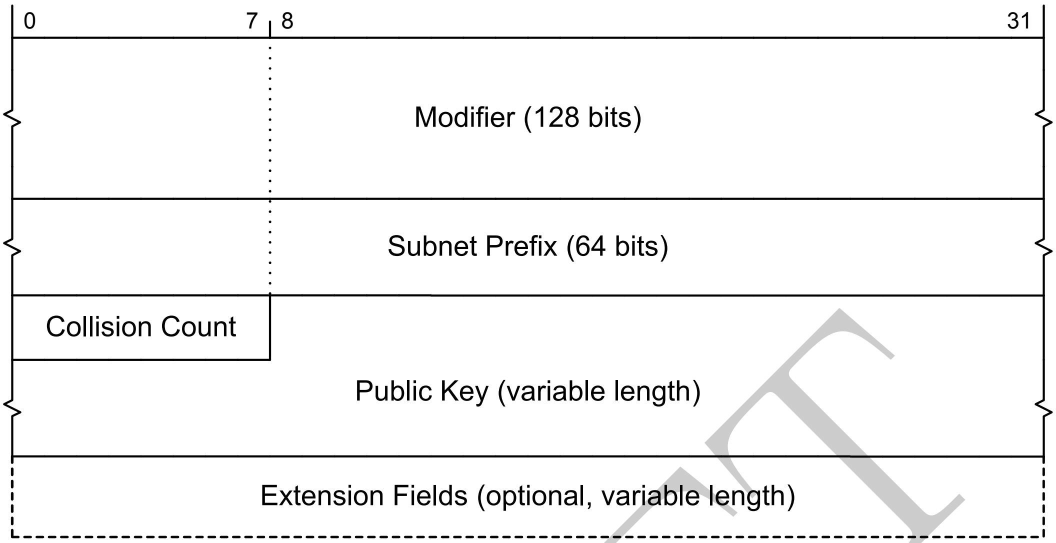

0
7
8
31
Modifier (128 bits)
Subnet Prefix (64 bits)
Collision Count
Public Key (variable length)
Extension Fields (optional, variable length)

Figure 6.3: CGA Parameters data structure format

Another interesting detail we can spot in the CGA specification is as follows. In a conformant IPv6 interface ID, only 62 bits are available for stuffing with auxiliary data because the other two are the **U/L** and **I/G** bits controlled by the Modified EUI-64 format from Section 2.7. If it were a plain 62-bit hash value, a brute-force attack on such an ID would have complexity of about 2^{62} values. Considering Moore's law, it can't be expected to stand up to the challenge for too long. To counter this threat, the most significant three bits in each CGA interface ID contain a special field, known as **Sec**. The CGA protocol makes ingenious use of this 3-bit field to provide for gradual adjustment of the resistance to a brute-force attack at the cost of more complex and CPU-consuming generation of the interface ID [181, Section 7.2].

The **Sec** trick is akin to brute-forcing a hash function pre-image. Namely, the task is to find such a CGA modifier value that will turn $16 \times Sec$ high-order bits to zero in the hash value $H(\mathit{modifier} \mid \mathit{public\ key} \mid \mathit{etc})$. CGA modifiers not meeting this requirement are considered invalid. Thus the brute-force attack complexity is increased to $O(2^{59+16\times Sec})$ [181, Section 7.2][182]. At the same time, generating a CGA becomes an $O(2^{16\times Sec})$ task; but CGA verification can still be performed in an $O(1)$ time because the only extra action required will be to calculate the hash value and ensure its $16 \times Sec$ most significant bits are zero.

The original CGA protocol suffers from one serious shortcoming: a fixed fingerprinting hash function, SHA-1. With hash function weaknesses actively researched by both mathematicians and hackers, it shouldn't be ruled out that any given hash function will eventually be cracked. If the SHA-1 fortress ever falls and its reversing becomes feasible, the CGA protocol will have to be extended to avoid a flag day. To see why it is so, consider the following naive alternative: Suppose new implementations use a stronger hash function such as SHA-3 on output but still try both functions on input because they can't know for sure which function a particular CGA was created with. Then a variant of downgrade attack, aka "bidding-down attack", will be possible as follows: For a trusted CGA created with SHA-3, such a key pair is found that its SHA-1 fingerprint has the same value as the authentic SHA-3 one. If this computation is successful, a backward-compatible implementation will accept the rogue signature as valid because it will try SHA-1 along with SHA-3.

It is worth pointing out that, in order to compromise a CGA, a second pre-image of the SHA-1 hash value is needed, while only finding a collision of this hash function is deemed practical today. For an introduction to different types of attacks on hash functions, see [183] and [184].

To be proactive and not to wait until SHA-1 is cracked, the CGA protocol has already been altered to accommodate different hash functions [185]. Unfortunately, this CGA extension can be a textbook example of the trouble to come about when the initial version of a protocol lacks flexibility in some respect. The CGA hash function solution from [185] isn't too elegant and, which is worse, overrides a protocol already published. Namely, the **Sec** field gets a new, fairly confusing interpretation. The good news is that CGA isn't out of the experimental phase yet, and so the toll of such a change can be relatively low.

CGA is among the mechanisms requiring an IPv6 interface ID length of 64 bits by protocol design rather than by policy. Although this is at odds with the original thesis from [21, Section 2.5.4] that the IPv6 interface ID length be variable for architectural purposes, it could be predicted from the outset that a trade-off would be necessary at least in some cases. This is how it works out for CGA: On the one hand, the existence of the CGA protocol doesn't preclude a different interface ID length on a subnet using no CGA; on the other hand, when CGAs are actually deployed, one has to stick to 64-bit interface IDs.

Great, now we have a cryptographic solution that satisfies the $O(N)$ requirement to its configuration overhead, where N is the number of neighbor

Table 6.1: Addresses claimed by ND message types

ND message type	Address claimed
NS from an unspecified Source Address (DAD)	Target Address
NS from a non-zero Source Address	Source Address
NA	Source Address = Target Address (not applicable to ND proxy)
RS	Source Address (unless unspecified)
RA	Source Address
Redirect	Source Address

nodes involved. So it is time to incorporate it into the ND protocol.

To validate a signed ND message, its recipient will need to consider the following: the basic ND fields, the signature, the public key, the other CGA parameters, and finally the IPv6 address claimed by this message. Which of the addresses shown in the message is claimed depends on the message type [186, Section 5.1.1]. Having made it through Chapter 5 of this book, we have sufficient knowledge of ND to work that address out for each ND message type as shown in Table 6.1.

ND proxy replies are shown in Table 6.1 as unsupported because a proxy can't know the private key of the node it responds on behalf of. Securing a Neighbor Advertisement therefore is possible only as long as its Source Address is the same as its Target Address [186, Section 7.4]. A method for securing ND proxy replies is in the works [187].

Having finished this section, work out how a host can trust the IPv6 address of the router that issued a signed Redirect. (Hint: *The router address can be trusted if it was learned beforehand from a manual configuration setting or a certified Router Advertisement.*)

In the first step of ND message validation, the recipient will need to verify that the public key presented in the message indeed has a fingerprint equal to the one embedded in the IPv6 address claimed by the message, assuming it is a CGA. Once the public key authenticity has been confirmed, the rest of the message data can be validated using the public key and the signature.

Consequently, a signed ND message needs to contain, apart from basic ND data, the following bits: CGA parameters, a public key, and a signature.

It will be natural to include them as special ND options. The rules for handling such options come as a central part of the **Secure ND** protocol, aka **SEND** [186].

> To be precise, the CGA parameters and the public key constitute a single ND option known as CGA option. The CGA option can be left out completely if the neighbor's public key is already known via a different trusted mechanism; in that case a signature alone will be sufficient.

The CGA mechanism as employed by SEND can be reliable only provided the trusted IPv6 address is selected by the authenticator node beforehand, by independent means. The signed ND message must be about the address selected so. Only then the entire mechanism can rest on the solid foundation of the trustedness of the address. For example, when we *ping* an on-link IPv6 address, our local node will send a Neighbor Solicitation and look to receive a Neighbor Advertisement about no other address than the one we are trying to ping. So at least unicast address resolution fits in the CGA model.

> At the same time, nothing prevents an attacker from generating a new CGA, along with a key pair, and "poisoning" the victim node's NC cache, e.g., through a signed Neighbor Solicitation with an SLLA option. Still, the rogue NC entry will do little harm unless the victim node actually tries to speak TCP/IP to the fake CGA or use it as a next hop. It is the essence of the CGA security model that any communication needs to use trusted addresses.

> Although CGA can also be used in protocols other than SEND, great care must be taken if doing so because that can enable cross-protocol attacks, e.g., where security data from one protocol is replayed in another protocol to gain unauthorized access to its resources [181, Section 7.4].

As for Router Discovery, it presents a challenge to CGA because a host can't know in advance what the router address is going to be, the whole point being to discover it dynamically using this subset of ND. So the use of CGA alone fails to rule out that some Router Advertisements come from rogue routers [188]. Only preventing a *known* IPv6 address from being snatched is within CGA powers.

What can secure Router Discovery is router certification. That is, each authorized router is to get a certificate issued by a trusted certification authority (CA), possibly through a chain of trust. The initial host configuration will then boil down to installing a copy of the trusted root certificate on each host. Having the root certificate, a host can rely on conventional crypto procedures to ensure that a signed Router Advertisement comes from a legitimate router.

The CA private key can be expected to be inaccessible to an attacker, e.g., locked up in a safe box. So the attacker can't create a new valid router certificate: she can only steal the certificate and the private key of a router, e.g., through breaking into its OS remotely. Should that happen, the attacker will be able to take over the router's identity, and that's it. As soon as the incident is detected, the compromised certificate will have to be revoked through a certificate revocation list (CRL), which is standard practice when dealing with digital certificates. A host therefore must check first of all that the router certificate hasn't been revoked. Isn't it a chicken-and-egg situation? The CRL is supposed to be downloadable from a central store, e.g., via HTTP or LDAP, which may require a router because the CRL store can be off-link. In turn, a rogue router can try to manipulate the CRL being downloaded through it. Fortunately, the CRL should be signed by the same trusted CA whose certificate is locally available to the host. The rogue router thus can't tamper with the CRL: all it can do is prevent its download altogether. Consequently, the host must stop using the current router if the CRL download failed. In other words, the host first selects a router tentatively, only to download the CRL, and promotes the router to usable status only if all of the following conditions are met:

- The CRL has been downloaded successfully.
- The CRL downloaded has a valid signature by the trusted CA.
- The current router's certificate isn't in the CRL.

Otherwise the host should log a security alert and try a different router learned from its RAs [186, Section 6.3.1].

These simple ideas are what the router authorization protocol under the SEND umbrella is based on [186, Section 6]. We won't discuss it in depth, pointing out instead that most of its gory details concern passing chains of trust within an auxiliary sub-protocol known as **Authorization Delegation Discovery** (**ADD**).

Unfortunately, ND is susceptible not only to straightforward attacks with rogue ND messages. Its state machine will react to other events as well, such as a transit IPv6 packet causing creation of an *INCOMPLETE* NC entry. A DoS attack therefore can be mounted where the victim router is flooded with transit packets addressed to different non-existent nodes of a connected subnet [178, Section 4.3.2]. A naive router implementation will just fill up all of its RAM with *INCOMPLETE* NC entries and crash. What can prevent this outcome is adaptive NC aging whereby *INCOMPLETE* entries are timed out at a rate depending on the cache utilization. A similar consideration can apply to a host if the attacker is able to start applications on it that send packets to a user-specified address.

6.4 Managing IPv6 Multicast

Conceptually, IPv6 multicast is no different from its IPv4 cousin because their lines of development have been matched for a long time. So, even if you have only a nodding acquaintance with IPv4 multicast, you still won't have a problem working through this section because there will be no references to IPv4-specific knowledge here. Rather, you will have a chance to catch up on this subject for both IP versions at once.

Formerly an exotic feature, multicast has come to be a fundamental mechanism with the advent of IPv6 and it can no longer be overlooked. So just bear with us through this section.

When busy "building" such an important IPv6 component as Neighbor Discovery back in Chapter 5, we would rely heavily on IPv6 multicast, taking it for granted that it could do its job for us. So now we should take time to ensure it will actually be the case.

While zooming in on IPv6 multicast, we would love to be able to assume that the underlying link-layer multicast is always functional and ready to serve. This would be a natural assumption in line with the general model of a network protocol stack adopted since the dawn of TCP/IP. However, it is particularly true for multicast that the devil is in the details.

So let's first refresh our memory on what levels of multicast support can be found in the link layer. There are at least four of them:

- First, there are link technologies providing native multicast, at least in theory. For example, an ideal *Ethernet* LAN will happily deliver a multicast frame to all members of the group as soon as we specify the right MAC destination address with the **I/G** bit set. If we move on to the practical plane, though, there will be a few caveats. In the historic shared-bus *Ethernet* the frame bits will be received by all stations at the physical layer, at least as many of them as needed to get on to the MAC destination address in the frame header, and the decision whether the frame is interesting will have to be made by the network interface card in the first place. For this reason a typical *Ethernet* NIC has a programmable multicast address filter, but its capacity is usually limited and if the host tries to listen to more link-layer groups than supported by the filter, all multicast traffic will have to be received by the NIC and passed up to the host software. A modern switched LAN is smarter than the shared bus as far as unicast traffic is concerned, but multicast delivery still can't be selective unless there are some mechanisms yet to be specified because the switches have no way of learning which ports the group members are on. We will revisit the latter problem in a little while and try to solve it. Until then, the switches have to flood each multicast frame to all ports in the LAN but the source port.

- The second group consists of broadcast LAN technologies having no multicast. A typical, if outdated, example of such a technology is *ARCNET*. Each *ARCNET* station had an 8-bit link-layer address. In addition, the address of all zeros meant broadcast. However, there was no such thing as an *ARCNET* multicast address. To make up for that, any IP multicast address, when operating over *ARCNET*, had to be mapped to the zero *ARCNET* address in the hope that all stations would apply an IP-level filter rather than blindly consume the packet. Such "multicast" had no selectivity at the link layer.

- The third group comprises point-to-point links where all addressing is implicit, e.g., PPP. By sending a packet over such a link, the local node effectively tells, "This packet isn't for me." Then the link layer has no options but to conclude from that, "If it isn't for you, it must be for the other node." So in this case, too, the multicast packet gets delivered to a node that may or may not be a member of the multicast group addressed.

- Finally, there are NBMA links, in which multicast is usually crippled if not at all impossible. We discussed them at length in Section 5.2. The best a node can do when on such a link is fan the multicast packet out to all of its direct neighbors, effectively treating the NBMA link as multiple point-to-point ones. That some members of the multicast group may never receive the packet can't be helped unless the link is

> converted to broadcast semantics with an adaptation protocol such as LANE in ATM or VPLS in MPLS. At the same time, the packet can be received by some non-members because the source node has no idea which of its neighbors actually are members of the destination multicast group. Although this mode of operation can hardly be regarded as real multicast, it still can be useful, e.g., for router discovery since the router is supposed to be a neighbor of each host of the link, as was discussed in Section 5.2—back then we mentioned this funny mode under the name of "pseudobroadcast".

The bottom line is this: If a link supports multicast, it will do its best to deliver a multicast packet to all members of the destination group, but it is allowed to sacrifice selectivity and deliver the packet to some non-members as well. Therefore *a node must filter incoming multicast traffic by the destination address at all practical layers.* In reality, the filtering at the link layer can't be fully relied upon because of the link-layer multicast addressing often being rudimentary, and so the fine filtering, to pass only packets relevant to this node, is up to the IP layer.

In the above requirement for a node to filter incoming traffic, we spoke of multicast because it is the subject of this section. In fact, the same requirement applies to unicast traffic as well. In particular, a host must not assume that any incoming frame or packet is for it. Foreign packets may appear at the host interfaces by mistake, e.g., due to outdated routing information in the network. So, for the sake of reliability and security, every incoming packet's destination address must be verified by the stack before the packet can be consumed by the host. We do the same thing in our everyday lives when we make it a rule to check the address on an envelope before opening it even if it came in our private mailbox.

Now that we have some idea of what is going on at the link layer with respect to multicast, we can start moving our focus up the stack. The next thing to encounter on this way will be the software interface between the link and IP layers. By the well-known model of encapsulation, a multicast frame is turned into a multicast packet by merely stripping off the link-layer framing such as *Ethernet* header and trailer, which we call "decapsulation".

Strictly speaking, a unicast packet may as well be found in a multicast frame received. Some non-TCP/IP protocol families even make use of this trick to skip explicit resolution of unicast addresses. As was mentioned in Section 5.2, one of them is ISO.

At the same time, going the other way around, i.e., encapsulating a multicast packet into a frame, can require a bit more effort than decapsulation if the link supports an explicit addressing of multicast groups. What link-layer destination address will need to be specified in the frame header? The answer to this is given by the link type-specific multicast address resolution rules, which are to be local and boil down to a calculation by what was said in Section 4.1.1. For instance, over *ARCNET* any IPv6 multicast address will map to the same broadcast address *0* [172, Section 7] and over *Ethernet* it will be the bit-masked address *33-33-XX-XX-XX-XX* [121, Section 7]. Even in the latter case the mapping isn't one-to-one, but it is OK as long as a rigorous filtering is applied on input, as just discussed.

This can be another way to view the issue of multicast addressing selectivity: The loss of information when mapping an IP multicast address to a link-layer one calls for a mandatory filtering of incoming packets by their IP destination address. This equally applies to IPv6 and IPv4 since IPv4 multicast resolution procedures would also lose bits, e.g., when mapping to the same zero *ARCNET* address [171, Section 4.3] or to the MAC-48 *01-00-5E-XX-XX-XX* [64, Section 6.4].

With our dive down the stack finally over, we can come up back to the IPv6 layer. The problem of IPv6 multicast can be divided in three distinct tasks, which aren't completely new to us because they have unicast counterparts:

- on-link sending;
- receiving;
- routing.

The on-link sending of IPv6 multicast is mostly clear to us: The sending node, be it a host or a router, somehow selects an egress interface for the packet to send and passes the packet down to the link-specific handoff procedure, which in turn computes the link-layer destination address from the IPv6 multicast destination address if required and—off we go!

Receiving IPv6 multicast can be a bit more complex as it requires some preparation. When discussing multicast-based mechanisms for IPv6, we would often say that some node needed to join group G on interface I. Now it is time we worked out what it really meant. First of all, the node has to program its IPv6 and link-layer input filters so as to let in packets sent to group G. Conceptually, this step can be as simple as adding address G to the list of groups listened to on interface I. Since multicast group membership is dynamic in nature, the reverse operation will also be required, that is, to leave group G.

> In IP multicast, sending and listening are decoupled, allowing a non-member to be sending to a multicast group. Only the reception path is affected by group membership.

On-link multicast assumed operational, it is time to go beyond a single link and consider multicast packet routing. Suppose there are off-link members in the multicast group a source is sending to. Will the source node need a multicast routing table or, at the minimum, a default multicast router? Will it have to distinguish on-link and off-link recipients, like when unicasting? All this hassle can be avoided altogether if the multicast router pretends to be just another group member and starts listening to the group in question instead of being an explicitly selected next hop for off-link members, as illustrated by Fig. 6.4. Still, such a router will only forward the packets received so, not consume them as a host would do, and so we should not call it a member of the group. Instead, it will be just a **listener**. The members of a multicast group are also its listeners, but not all its listeners are members.

For this scheme to be feasible, the multicast router has to learn somehow what groups to listen to and where to forward them. Any static configuration would be out of keeping with the dynamic character of multicast here. Suppose a router connects N links, $L1$ through LN. If the source sending to group G is on $L1$ while its listeners are on $L2$, the router is supposed, at the very least, to listen to G on $L1$ and forward its traffic to $L2$. The most crude solution would be for the router to listen to G on all N links and fan out each packet to $N - 1$ links, excluding the one it came from. This isn't too different from flooding by a LAN switch.

Is there a way to optimize this process? A LAN switch would learn which of its ports a node is on by looking at the MAC source address of incoming frames and assuming that the LAN path is symmetric. Thus the switch would go like, "If a frame from MAC source S came in via my port $P5$, the best I can do is forward frames with a MAC destination of S only into port $P5$ from now on." Alas, this trick isn't going to help us out with multicast because it is impossible to infer from the ingress interface of a multicast packet which links have *listeners* of that group on them. This problem stems from the listeners making no effort to reveal themselves while indirect criteria fail. First, a source sending to a multicast group isn't always that group's member. Second, a multicast address will never show up as the IPv6 Source

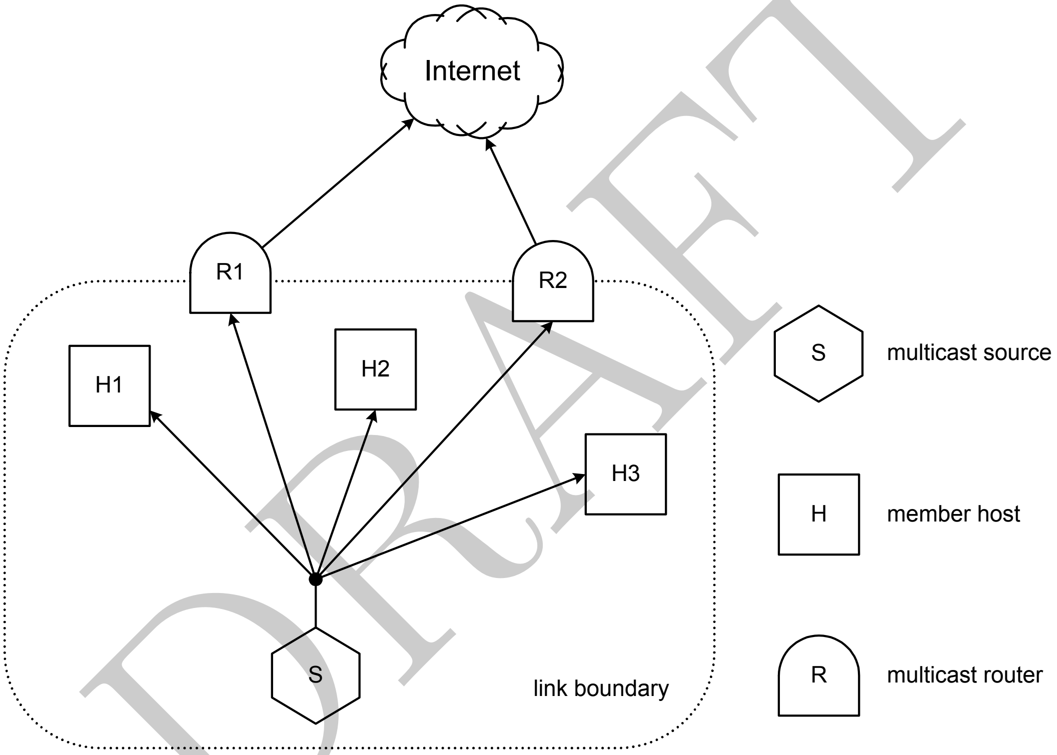


Figure 6.4: To a multicast source, routers look the same as hosts

Address of a valid packet. Third, not all listeners are members authorized to speak on behalf of the group.

> A LAN switch can learn from a multicast as well as unicast frame, but in any event the MAC source address will be unicast and so MAC address learning is limited to unicast addresses only. Therefore a LAN switch would face essentially the same problem when trying to manage multicast traffic at link level. The LAN switching considerations will be touched on, too, later in this section.

To make up for this lack of information, IPv6 multicast listeners will have to advertise themselves using a purpose-built protocol. Since it will essentially be yet another aspect of IPv6 control, it can be conveniently implemented within ICMPv6. Its name will be **Multicast Listener Discovery** (**MLD**).

At the same time, multicast sources will reveal themselves through just sending data packets, making a multicast source discovery protocol redundant. A multicast router will simply need to set up its interfaces so that they accept all multicast packets. For example, as long as a router is concerned only with IPv6 and *Ethernet*, it will need to accept any MAC frame whose destination address begins with `33-33`. Still, such background reception of all multicast traffic by the router doesn't automatically make it a listener in all possible groups. The router will be listening to a group's traffic only when it is ready to forward it on. This distinction is important because a true listener is supposed to advertise itself with MLD in addition to just accepting the packets.

> The IPv4 counterpart of MLD is IGMP. Its modern versions are based on the same principles and its message types play the same roles as their equivalents from MLD. Furthermore, the two protocols are virtually harmonized because the lines of development of IPv4 and IPv6 multicast closely match each other. In particular, IGMPv2 corresponds to MLDv1 while IGMPv3 mirrors MLDv2. What those version numbers mean, we are just about to see.

Let's start actual work on MLD by figuring out how much information about the listeners of group G on link L will be required for the link's router to do its job. Is it going to need a complete list of listeners' IPv6 addresses? When forwarding a multicast packet over link L, the router just hands a single copy of it down to the link layer, and the rest is done by the link layer mechanisms. This being the case, *the multicast router needs only to know if there are any listeners on the link at all, whereas their number or identity is irrelevant to the router.* Much of the discussion to follow will revolve around this simple point.

> Ideally, a multicast router needs to keep track of links connected rather than just of interfaces because some of its interfaces can end up connected to the same link and that may result in duplicate copies of packets being transmitted. The worst-case scenario will be an exponential multicast storm when the router receives multiple copies of each packet it forwarded and proceeds to forward them too. However, a means to configure such an interface-to-link map is implementation-dependent.

Another fact regarding MLD is that its scope will be just a link. So link-local scope should be a perfect fit for both the Source Address and the Destination Address of an MLD message. The reliability of data transmission within a link is usually satisfactory, and so MLD doesn't need to bother itself with reliable end-to-end delivery of its messages. Instead, just repeating MLD messages periodically will be as effective.

> For those who enjoy researching the history of TCP/IP, we will point out that the earliest version of IGMP [189, Appendix I], referred to by some as IGMPv0, did provide for reliable message delivery from a listener to the router. Later practice showed, however, that the reliable delivery wasn't strictly necessary, thus permitting to simplify implementations. The thing is, a reliable delivery protocol has to be stateful while periodic transmission of messages can be stateless except for a timer.

Our earliest draft of MLD can look as follows. Each listener sends **Report** messages about the groups it is interested in receiving, showing the IPv6 multicast addresses of those groups. A router can prompt listener reports by sending a **Query** message although a listener is also allowed to send an unsolicited Report whenever it deems that necessary. In order to avoid MTU-related issues at this early stage, let each Report message contain just one multicast address. Should a listener have joined multiple groups, just as many Report messages will be sent. Which party will be setting the pace in this scheme? Until there is a multicast router on the link, there will be little use for periodic Reports. Therefore let the router time its Queries while the listeners will be just responding to them with Reports.

To avoid concerted bursts of traffic on the link, the listeners should delay their Reports by a small random interval. Message retransmission intervals should also be randomized.

There still is a moment when a gratuitous Report is to be sent immediately; it is when a listener only starts listening to a group. Here we have one more action for a host to perform when joining group *G* on interface *I*: to send an MLD Report about group *G* over interface *I*. Since the link isn't expected to be 100% reliable, this Report needs to be repeated a few times just in a case.

What are the possible values for the IPv6 Source Address of an MLD Report? If the sending interface has already got a link-local address, it will be the winner. However, when an MLD Report is to be sent, the interface may have no such address, or even no address at all, e.g., if group *G* to be advertised is the solicited-node multicast group for a tentative link-local address (see Section 5.1 and Section 5.4.1 to refresh your memory on what those terms mean). We have found ourselves in a chicken-and-egg situation again: To join a group, a local address is required, and to get such an address, a group needs to be joined first. Fortunately, we have a stock solution to it: An MLD Report packet will be allowed to have an unspecified IPv6 Source Address, *::* [190][191, Section 5.2.13]. This is acceptable because the router won't track listener identities anyway, as we said earlier.

All solicited-node multicast groups being link-local, do they actually need to be reported in MLD? As will be clear in a couple of paragraphs, this is a very sensible question for us to discuss.

If MLD sticks to this scheme, the Query message will need no parameters at all and the Report message will have just one field relevant to MLD: the IPv6 address of the group reported. How will the router detect that all listeners of a group are gone when all of them have left the group? When that happens, no Reports about that group will come back after a Query. Since Query and Report messages can be lost, the router shouldn't rush to stop forwarding traffic to the group though: it needs to wait for a few more Query cycles. If even then no Reports about that group come through, it will be safe to conclude there are no listeners left on the link.

The last decision for us to make at this stage is regarding the IPv6 Destination Addresses of the Query and Report packets. The Query is supposed to reach all IPv6 nodes of the link because any of them may be listening to some multicast group. So the Query is sent to the All Nodes of the Link group, *FF02::1*. To avoid yet another chicken-and-egg problem, this group has to be made special, as in: it never needs to be reported with MLD. A link-local group, it needs no routing anyway.

An alternative solution could have been to create a dedicated IPv6 multicast group for all listeners of the link. However, it would have bought us nothing because multicast is implemented and used by all IPv6 nodes, e.g., for Neighbor Discovery, and so each IPv6 interface is a member of multiple groups. (List them as an exercise.)

An IPv6 router is permitted to forward a unicast packet with a link-local Destination Address back into the same link it came from; this won't violate the scoped address architecture from Section 2.4. At the same time, forwarding a multicast packet with any Destination Address scope back into its link of origin is a nice way to cause a loop. Moreover, should there be multiple multicast routers on the link, a multicast storm can be triggered this way, with the number of packet copies growing exponentially with each hop. To prevent this failure mode, the IPv6 **Hop Limit** is set to 1 in MLD packets. Consequently, GTSM protection (see Section 5.1) isn't applied to MLD.

The general rule will be that no traffic sent to a link-local multicast group is to be routed, and so the routers don't really need to know about such groups. If so, should link-local groups be excluded altogether from MLD Reports? In fact, no; they still need to be reported because MLD Reports about them are of interest to smart LAN switches "snooping" for the group topology in order to optimize multicast traffic. MLD snooping will finally be discussed a few paragraphs below, as we have long promised.

As for the interface-local multicast addresses (*FFx1:...*), they are relevant only within a particular interface and packets sent to them must never appear on the wire. Quite naturally, such addresses must not show up in MLD messages. The same applies to the reserved multicast scope 0.

As for the MLD Report, we can name as many as three candidates for its IPv6 Destination Address: the unicast address of the router that sent the last Query; the All Routers link-local multicast address, *FF02::2*; and. . . group G to be reported. The first option, the querying router's unicast address, isn't optimal because the other routers that may be present on the link won't receive the Report and the listeners will have to respond to each router individually. The routers already see each other's Queries, according to our draft scheme, and so they can try to elect a single **querier** that would be the "metronome" to be setting the pace in order to minimize the total number of Queries sent and so to reduce the overhead from MLD running on the link. We will push the issue of how a querier election can be performed onto our brain stack for now, but the optimization idea is quite clear: only one router, the querier, is to be sending MLD Queries while the Reports are received by all routers. The second candidate, All Routers, allows for such an optimization, but it does so at the price of bothering pure unicast routers with MLD Reports. Finally, using the same group G the Report is about for its IPv6 Destination Address will get the Report through to all multicast routers because they have to be receiving all multicast packets from the link anyway, as we discussed earlier. In addition, the Report will be received by the other listeners of group G, which may be of help in further optimization of MLD Report sending. Based on this rationale, let an MLD Report be sent to the IPv6 multicast address it is about—at least in the first version of MLD we are coming up with.

The additional optimization to MLD Report sending is as follows. Since multicast routers don't keep track of each listener present, a single Report per group listened to will be sufficient. While a listener delays its Report about group G for a random interval to avoid a traffic spike, it pays attention to other listeners' Reports about the same group that may come through if the other listeners' delays were shorter. If such a Report is received, another listener has already advertised this group and sending the pending Report within the same Query cycle would be redundant. This trick, known as MLD Report suppression, has a downside though, to be discussed in the context of MLD snooping.

The simple scheme employing just two message types, which we have taken the liberty of referring to as "our early prototype" of MLD, is the basis of IGMPv1 in IPv4 [64, Appendix I]. In it, querier election is left up to a multicast routing protocol.

Our work has so far resulted in a protocol in which a multicast listener would simply tell its current state by reporting all of the groups it is interested in receiving. A router can solicit such Reports with a Query, but it can't affect their contents. In essence, MLD was born stateless, as in: A protocol message will affect neither the long-term state of its recipients nor the contents of the subsequent messages. So to speak, MLD parties aren't conducting a real conversation; rather, they are making unconnected remarks. Consequently, MLD Reports can be received and correctly interpreted by a third party snooping on the link while MLD Queries can be disregarded by the third party altogether because they contain no useful information for it.

We will see later in this section that the reception of Queries along with Reports can help to provide a finer control over the MLD timers. That being said, an MLD Report can be fully understood without knowing the triggering MLD Query. Moreover, an MLD Report can be unsolicited, e.g., when the reporting listener is just joining the group.

This feature of MLD enables optimization of link-layer multicast in addition to IPv6 multicast. For example, a smart *Ethernet* switch can watch MLD Reports coming in through its ports and thus discover listeners on a per-port basis. Granted, this trick smells of network stack boundary violation because a basic Layer 2 device will have to pick MLD messages from among IPv6 packets from among all *Ethernet* frames and then parse and analyze them. This will have to be followed by mapping IPv6 multicast addresses to their MAC counterparts because the switch manages traffic by MAC addresses in the first place. Still, this trick should be quite within the capabilities of modern LAN switches, whose number-crunching power is well above what the computing pioneers could dare dream of. Because the LAN switch will be monitoring MLD messages in spite of its basic role, the technique is known as **MLD snooping**.

The same trick, when applied by LAN switches to IPv4, becomes IGMP snooping.

Note how the IP multicast address resolution rules make IGMP/MLD snooping possible. Only because they are local and boil down to a bitwise operation, a LAN switch has no trouble mapping an IP multicast address to the corresponding MAC address. If it weren't the case, the Layer 2 switch would be unable to make use of the Layer 3 group membership information from an MLD or IGMP Report, since link-layer switching is ultimately done based on MAC addresses.

Controversial as it can seem from an architectural point of view, IGMP/MLD snooping has become the de facto method of choice when it comes to managing IP multicast traffic by link-layer means. Even well-known protocols have to reckon with it now. In particular, IP multicast listeners have to report any link-local groups except *224.0.0.1* and *FF02::1*, which is not for routing purposes but to help the LAN switches learn their topology through the snooping process.

> Can you come up with a trick for a LAN switch to discover IPv4 and IPv6 listeners of the special All Nodes groups, *224.0.0.1* and *FF02::1*, given that neither group is to be reported in IGMP/MLD? In other words, how can the switch detect as early as possible that there are IPv4 and/or IPv6 nodes on a given port? (Hint: *ARP with a Protocol Type of* `0x800` *or BOOTP/DHCPv4 indicates IPv4; ND or DHCPv6 indicates IPv6.*)

With a minimal prototype ready for MLD, let's invest some time in optimizing it. First of all, we need to deal with a serious performance problem we created. When sending a Report about group G to the multicast address G, the listener expects the Report to be received and handled by all of the multicast routers present on the link, not only by the other listeners of group G. This distinction is of particular importance when the group is reported on that link for the first time and the routers have never heard of it before. We said above that a multicast router would need to program its input filters to the reception of all multicast packets from the wire. But, in order to notice the Report message, the router would also have to perform deep processing of nearly each incoming multicast packet, all the way down its IPv6 header chain, because almost any such packet can turn out to be an MLD Report. This would be well beyond what an IPv6 node can be expected to be doing because such a node, be it a host or a router, should look past the main IPv6 header only if the IPv6 Destination Address is recognized as its own.

> Strictly speaking, IPv4 was different in this respect because an IPv4 router had to scan through all IPv4 options in any event.

> In IPv6, there is just one exception from this rule: Hop-by-Hop Options from Section 3.3.2. They are going to help us out now.

As we know well, the normal way for an IPv6 node to pick out relevant packets is based on the IPv6 Destination Address. If this address is unicast, it needs to be found on any interface of the node; and if it is multicast, the corresponding group needs to have been joined on the ingress interface of the packet. Otherwise a host will just drop the packet while a router will try to forward it. For example, a router will consume and process an ICMPv6 Echo Request, a BGP TCP segment, or an OSPF message only if one of its own addresses appears in the Destination Address field of the main IPv6 header of the packet received. But an MLD Report message is really singular in this respect: From a router's point of view, it is sent to a random, unpredictable multicast address! Unless some kind of cue is provided along, a multicast router will have to dig deep into nearly each multicast packet received, first to work out if it is ICMPv6 and then to tell if it is MLD.

> Recall what we learned from Section 3.1: Unlike in IPv4, the upper-layer protocol payload type may not be available right from the main header of an IPv6 packet.

Such a cue, if provided, has to be inserted by the source of the MLD packet and it should be easily accessible by a router. What can it be? Now it is time to recall that, of all Next Hop values, one was set apart in Section 3.3.2 to indicate a Hop-by-Hop Options Header, to be scanned through by each node in the packet's path through the network. The distinctive Next Hop value in question is 0 (zero), its feature being that it can appear only in the main IPv6 header for the sake of easy access by routers.

A Hop-by-Hop Options Header thus can be used to mark "strange" packets defying the IP addressing conventions. Indeed, routers have to check for this extension header anyway and the cost of this check is low because it is limited to the main IPv6 header instead of spanning the entire extension header chain. We haven't used Hop-by-Hop Options up to now only because there has been no job for them, but now we have got one. To create a sufficiently general mechanism, let's introduce a Hop-by-Hop Option that will tell a router that the payload of the packet might be of interest to it although the packet isn't addressed to it explicitly. For its role, the new option will be known as **Router Alert** [192].

To define a new IPv6 option, its type code needs to be constructed first. As we remember from Section 3.3.2, the most significant three bits of the type are to reflect the option's properties. One property is whether the option can be safely ignored, and this is the case for Router Alert. The other property is immutability, and we can see no good reason for a transit node to fiddle with a Router Alert. This combination of properties is encoded as `000`, and so the type value for this option needs to come from the range 0–31. Its actual well-known type value is 5.

Will the Router Alert option need to carry additional data? Its very presence in a packet is a clear signal to a router, but the router will be helped even better if the option tells what kind of control message is to be found in the packet. In essence, it is going to be a protocol taxonomy from a different angle: If IP protocol values to be found in Next Header fields focus on encapsulation, the Router Alert option will emphasize traffic control means. Each control protocol will get a code value from the corresponding

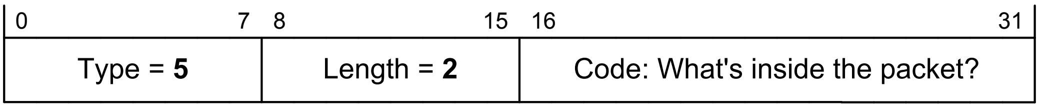


Figure 6.5: Router Alert option format

registry [193], thus enabling the router to quickly decide if the packet is interesting without having to work through the extension header chain to find out the payload protocol type. MLD was the first to come after such a code and got the sweet value of zero. For consistency, *let all MLD messages carry a Router Alert option with a code of* 0.

To see why yet another protocol registry was needed, think of IPv6 packets as pies and muffins: A Router Alert option can not only mark pies out among mere muffins but also tell what each pie's filling is, which makes a lot of sense because what a foodie cares about is the taste, not the recipe.

Discuss the threats to the security and stability of a network stemming from the use of Router Alert [192, Section 4]. How can they be countered or alleviated?

The code field in the Router Alert option is 2 bytes long, as shown in Fig. 6.5. What needs to be added to its specification to make it usable? (Hint: *It is an unsigned field in network byte order.*)

Using Fig. 3.8 and Fig. 6.5 as a reference, build a Hop-by-Hop Options Header containing *solely* a Router Alert option for MLD; fill in all of the fields. Can it be done exactly as requested? If not, how can it be helped? (Hint: *Padding will be needed as well, e.g., two Pad1 options or a single empty PadN option.*)

On to the next optimization. We still have on our brain stack the task to come up with a **Querier** election protocol. Such a protocol is needed to allow the multicast routers of a link to elect one among their number to be sending MLD Queries while the others are just silently receiving MLD Reports. Here is an elegant approach to this seemingly non-trivial problem: Let the router with the numerically smallest IPv6 address be the Querier.

An ordered bit string, an IP (v4 or v6) address can be regarded as a binary representation of an unsigned integer number. Thanks to positional notation being unambiguous, there is a one-to-one mapping between this binary representation and the inherent numeric value of the IP address, which guarantees that different IP addresses will never have equal numeric values. Precisely speaking, positional notation may be ambiguous due to leading zeros, but this caveat doesn't apply to an IP address owing to its fixed length. Were IP addresses to have a variable length, we would have to specify if leading zeros are significant, e.g., whether `1` and `01` are distinct addresses or just spellings of the same address. Quite obviously, the numeric value of an address is unique only within one IP version or the other but not across the versions.

We have already taken care in this section that the multicast routers on the same link will be receiving each other's MLD Queries. Initially, each of the routers will start sending its Queries from a link-local address known to be unique within the link thanks to DAD from Section 5.4.1. If, while doing so, a Query from a smaller IPv6 address is received, there is a better candidate for the Querier role and our router should stop sending its own Queries for an interval longer than the standard Query period. The end result will be that the best candidate will be suppressing Queries from the other routers with its own Queries. The other routers thus will be **non-queriers**. The mathematical basis for the election procedure is that the relations of "less than" and "more than" are transitive over a set of integer numbers: $A > B, B > C \Rightarrow A > C$. Thanks to this property, an election will always converge as soon as all of the routers receive all unsuppressed Queries from their competitors. If the current Querier goes down later, the timers will fire in the non-queriers, they will resume sending Queries, and it will be election time again, resulting in a new Querier elected. Here we have a simple and robust election mechanism.

In general, such an election can be conducted based on any unique IDs and any transitive relation over them.

In case there are multiple link-local addresses on a multicast router's IPv6 interface, for election purposes the router will need to stick to the one it would be sending its own Queries from on that interface unless suppressed.

Another detail to optimize is how soon a router can stop forwarding traffic to a multicast group with no listeners left on the link. The opposite transition, to start forwarding to a group, is immediate because each new listener sends a Report about the group first thing. By contrast, no cessation of listening is indicated yet, and so the router has to infer that there are no more listeners of the group when no Reports have been seen about it for a while. How long can it take? A Query, as it is now, is a quite expensive operation because it is first received and processed by all IPv6 nodes of the link and then responded to by each listener using as many packets as one per group. Therefore the Query sending period has to be rather long, such as a minute or two. So it can't take less than several minutes to detect the departure of all listeners from a group.

To speed this process up, let a listener send a message of a new type, **Done**, when ceasing to receive a group. Very much like a Report, the Done message body will need to contain the IPv6 multicast address of the group in question. And what Destination Address will suit the Done message the best? It could be the group's address itself, to match the Report, but the Done message isn't really interesting to the other listeners. In addition, it can be reasonable to expect that there will be fewer routers than listeners on an average link. So the Done message will be sent to the All Routers link-local address, *FF02::2*.

Now suppose the multicast routers of the link have received a Done message about group G from a listener. Since the routers neither track listeners individually nor even count them, the Done message only tells the routers that the number of listeners has reduced. A natural reaction to such a message will be to verify ASAP if there are any listeners left in the group. To do that, the Querier needs to send a few Queries over the link with short pauses between them.

If simple Queries were used for this purpose, they would trigger a whole storm of information beside the point. To avoid it, the Query message format has to be extended to allow for a more specific query concerning just one group. First, such a specific Query needs to be sent to address G instead of All Nodes of the link. Second, address G needs to be specified in the body of the Query message. Thus there will be a Multicast Address field in the MLD Query message format. Let it contain a zero value to indicate a **General Query** about all groups at once, or a valid multicast address if it is a **Multicast Address Specific Query**.

A small yet important detail of this mechanism is its interaction with querier election. What should the current Querier do if a fast check is in progress and all Queries haven't been sent yet but a foreign Query comes through from a smaller IPv6 address? For simplicity and stability, the old Querier will always run the fast check to completion before becoming a non-querier. This behavior won't hurt anything because querier election is just an optimization and there will be no harm from two routers sending Queries for a little while.

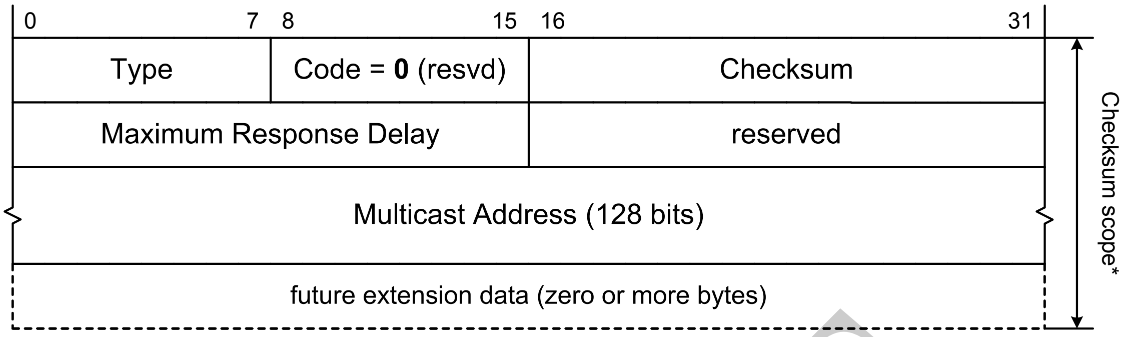
Our final touch to the fast check procedure will be regarding coordination between the Querier and non-queriers. How can the non-queriers know that a fast check is under way and the group timer needs to be set to a shorter interval? By our plan, non-queriers still receive all the Queries and all the timing is to be done by the Querier. So let the Querier tell the non-queriers the updated timer interval to be used after each Query. It can be conveyed as an integer field in the Query format. The listeners can also benefit from this information when generating a random delay value for their Reports because so the delays can be distributed over the entire range from zero to the timeout. For this reason, the field is known as **Maximum Response Delay**.

Non-queriers are to respect **Maximum Response Delay** only if it is a Multicast Address Specific Query. General Queries don't affect the group timers.

Ideally, listeners need to allow for the link transmission delay when generating a Report delay value based on **Maximum Response Delay**. Otherwise the Report may come late.

Now the wire formats for the MLD messages can be specified. Since all of the message types introduced are going to contain a Multicast Address field, they can share just the same common format inheriting the basic ICMPv6 features, as shown in Fig. 6.6. Of course, **Maximum Response Delay** will contain meaningful data only in a Query but not in a Report or Done message. Such a unified message has a fixed length of 24 bytes. What if extra data is found in an MLD packet following the fixed message? Let's not prohibit this case because it can be useful for future extensions. For now, such extra data is to be ignored in incoming MLD packets while outgoing MLD packets will have no such data after the fixed part.

What we have "built" up to this point can be released now as the first standard version of the Multicast Listener Discovery protocol, **MLDv1** [194].



*) Keep in mind that Checksum also protects the pseudo-header.

Figure 6.6: MLDv1 message format

> The same principles and mechanisms are at the core of IGMPv2 in IPv4 [195].

> It can be interesting to note that the ICMPv6 Type values assigned to MLDv1 messages are smaller than those of ND messages. This will make sense if we keep in mind that MLD precedes ND. E.g., a solicited-node multicast address first needs to be reported with MLD before it can be used by ND.

We have taken time to review and refine various protocol details for MLDv1, so let's conclude the work by compiling in Table 6.2 a list of its major features side by side with those of our unoptimized "zeroth version", which in fact mimicked IGMPv1 in IPv4. This way the progress made will be clearly seen.

Now, can we do even better in the second version of the protocol, to be known as **MLDv2** [191]? We bet we can! As proof of our potential, let's roadmap the following improvements right away:

1. The central object of the MLDv1 wire format is a multicast group. In particular, an MLDv1 message can concern either a single group or all of them, and the mechanism for redundant Report suppression can do no better than reduce the number of Reports to one per group. Thus the overhead from running MLDv1 can be expected to be proportional to the number of active groups, which can be quite large, especially on a backbone link populated mostly by routers, each forwarding traffic to many different groups. At the same time, the number of listeners

Table 6.2: "MLDv0" and MLDv1 feature chart

	"MLDv0"	MLDv1
ICMPv6 message Types	Query (130); Report (131)	Query (130); Report (131); Done (132)
Query variants	General Query	General Query; Multicast Address Specific Query
Report structure	One group per message	One group per message
Destination Address for Report	Group reported	Group reported
Querier election	Protocol unspecified	By IPv6 address
Redundant report suppression	Yes	Yes
Group timer coordination	No	Yes

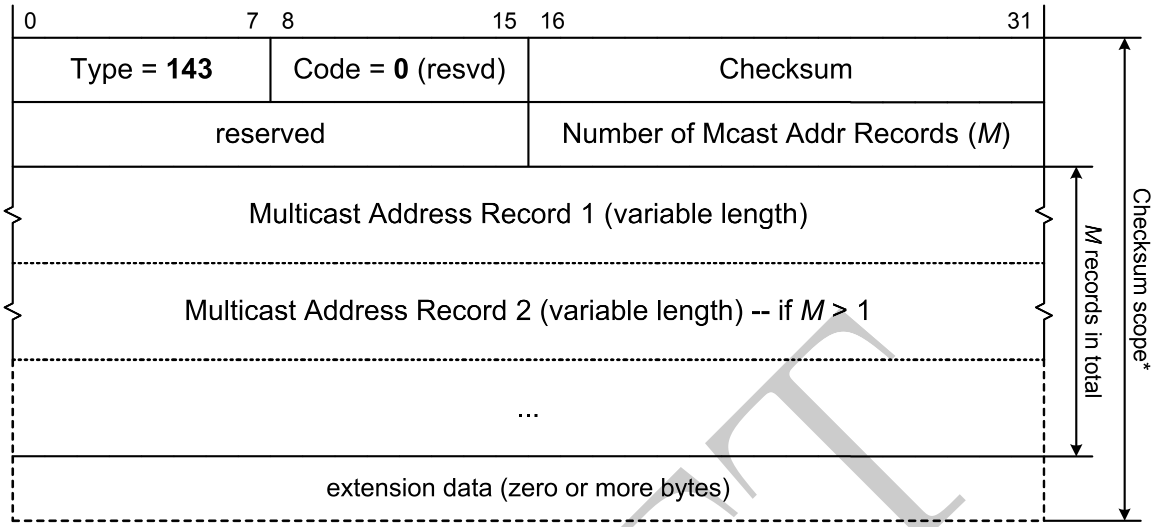
on one link isn't going to be off the scale for purely practical reasons. Therefore the MLDv2 wire data should focus on a listener instead of a group. In particular, a single MLDv2 Report message will be able to carry info about multiple groups at once.

2. MLDv1 supports only the most basic of the multicast addressing modes from Section 2.9, namely, ASM. Additional support for SSM and SFM will be a must for MLDv2.

3. The link-wide coordination mechanism is still rudimentary in MLDv1, conveying only Maximum Response Delay. Other important parameters such as the query interval and the retransmit count are to be synchronized as well.

> In IPv4, the same problems are solved by IGMPv3 [196].

The first improvement can be implemented through a more elaborate format for Report messages in MLDv2. Let it contain multiple records, each about one multicast group, as shown in Fig. 6.7. That makes us reconsider the choice of IPv6 Destination Address for such a message. In MLDv1, it could be the multicast address reported because a Report was about just one such address. And now that an MLDv2 Report is going to be about a variable number of different groups, we have no option but to have it sent to a new well-known multicast address, namely, All MLDv2-Capable


0
7
8
15
16
31
Type = 143
Code = 0 (resvd)
Checksum
reserved
Number of Mcast Addr Records (M)
Multicast Address Record 1 (variable length)
Multicast Address Record 2 (variable length) -- if M > 1
...
extension data (zero or more bytes)
M records in total
Checksum scope*

*) Keep in mind that Checksum also protects the pseudo-header.

Figure 6.7: MLDv2 Report format

Routers, *FF02::16*. Then the other listeners won't receive such a Report and the redundant Report suppression will no longer work. Its importance is already low in the listener-centric protocol, so that we can just give up on it altogether in MLDv2. Let each MLDv2-capable listener speak for itself and pay no attention to the other listeners. This will also simplify a listener implementation.

Another valid reason to give up on Report suppression has to do with MLD snooping by smart LAN switches [191, Appendix A.2]. In a simplest case, such a switch will forward a multicast frame only into the ports from which MLD Reports about that group have been heard. But if the Report was completely suppressed on some port, the switch won't forward the frame into that port because it has no idea there are listeners on it. To work around this problem, the switch just plays another trick and blocks the forwarding of MLD Report frames into those ports where there seem to be no multicast routers [142, Section 2.1.1]. The same trick can be applied to IGMP. Needless to say, stacking one trick on top of another can only result in hard-to-troubleshoot problems on the LAN.

A related problem concerns multicast routers connected to a smart LAN switch. As we said, such routers are expected to be receiving all multicast traffic from the link in the background, without actually joining the groups. Otherwise they will be, e.g., unable to receive MLDv1 Reports. Unfortunately, the switch won't be able to reliably detect all of the routers by their MLD Queries because non-queriers remain silent most of the time. To solve this problem in a straightforward manner, a simple protocol for **Multicast Router Discovery** has been proposed [197].

Trying to put information about multiple groups in one Report message can result in the link MTU exceeded; at the same time, IP-level fragmentation would be as undesirable for MLD messages as it is for all control traffic due to protocol stability considerations. To work around this problem, a single long MLDv2 Report needs to be made equivalent to a sequence of shorter Reports, each carrying a subset of the original multicast group list. This will enable an MLDv2-capable listener to report even a large number of groups in reasonably sized messages without fragmentation or segmentation as such, through just controlling the number of groups per Report message [191, Section 5.2.15].

The second improvement planned concerns support for the SSM and SFM addressing modes from Section 2.9. It will clearly be a non-trivial task because, when in either of these modes, each listener can potentially make a rather complex request to the routers, such as, "For multicast address so-and-so, these sources are to be allowed while those need to be blocked". Under these conditions, the challenge for a router will be to satisfy all the requests at once to the fullest degree achievable while spending as little of its own resources as possible.

Since a listener is supposed to filter incoming traffic in any event, letting a multicast packet from an unwanted source into the link will cause no problem except for wasting a bit of the link's resources. By contrast, blocking a source on the router by mistake will make an authorized communication impossible. For this reason, the router may only err on the permissive side. To be effective, this decision has to be backed by a consistent model for source-based filtering of multicast adhered to by both the listeners and the routers. So let's brace up and start drafting such a model.

First we should point out that, as far as multicast routers are concerned, there isn't much difference between SSM and SFM. It is up to a host to take into account that in SSM different source addresses S in the composite address (S, G) mean different channels that can be handled by completely different applications. As for a router, it only needs to optimize multicast traffic. Thus, should the host request reception of two SSM channels, (S_1, G) and (S_2, G), which differ only in the source but not in the destination, the router can just unblock sources S_1 and S_2 for group G while blocking the rest to save the network resources. The same approach will work for SFM.

> The fine details of managing SSM with MLD are discussed in [198].

The most general problem statement can be as follows: A router needs to allow a certain *set* of source addresses $\$$ to send to the given multicast address G while blocking the rest of the source addresses. So the natural language to express our filtering model in will be that of elementary set theory. The model will be concerned only with source addresses because each group will have a separate filter independent of other groups'. Let's refer henceforth to the *allowed* address set $\$$ as the master set because it will be the essential meaning of the filter according to the problem statement, even though there can be different ways to encode its information in the router's memory.

> For example, in a few paragraphs we will come up with an alternative encoding: the *blocked* address set.

In the set theory lingo, the so-called universe of the present problem, that is, the comprehensive set of all elements to ever consider, is just the IPv6 address space. It is finite, if very large, and so any source filter can be specified, in principle, by just enumerating all source addresses to allow. In particular, ASM would be implemented as a filter containing *all* IPv6 addresses; and leaving the group completely could be equivalent to applying a filter with no addresses in it: an empty set. Should there be multiple listeners with their requirements formulated as sets of source addresses to allow, the router will just need to build the union of those sets in order to satisfy all of the requirements at once, never blocking an authorized source: $\mathbb{S} = \bigcup_i \mathbb{S}_i$. Looks good on paper!

Consequently, MLDv2 won't need a separate Done message because leaving a group can be signaled using an empty source set.

We don't try to exclude bad source addresses such as multicast ones from the master set or the universe because we assume that packets get properly validated at an earlier stage. Thanks to this assumption, our set-theory discourse is simplified.

But what if an SFM listener requests just a few sources blocked and the others allowed? In this case the master set will be equal to all IPv6 addresses but the blocked ones, that is, to the blocked set's complement.[6] It isn't hard to see that N source addresses blocked will result in a master set of $2^{128} - N$ source addresses to allow—a completely off-the-scale number as long as N is small! The ASM mode, in which $N = 0$, still could be encoded as one special case; but full-blown SFM, which relies on selective blocking, would require all of the $2^{128} - N$ addresses be listed in the filter. Needless to say, such a colossal filter can be neither stored nor communicated while in its master form. So we have to resort to some tricks, theory-backed nevertheless, to overcome the filter size issue.

The most useful trick is to be learned from the selective blocking scenario. As long as the master set size is close to either of the extremes, none or all, the filter can be encoded tersely as follows. If the master set includes just a few addresses, they can be listed verbatim in the filter, to be allowed. On the other hand, if the master set contains almost all IPv6 addresses, it will be sufficient to list the addresses left out, i.e., blocked, because there is a one-to-one mapping between a block list $\overline{\mathbb{S}}$ and the master set $\mathbb{S}$ it represents: $\overline{\mathbb{S}} = \setminus\mathbb{S}$ and $\mathbb{S} = \setminus\overline{\mathbb{S}}$, where $\setminus\mathbb{S}$ denotes the complement of set $\mathbb{S}$ with respect to the task's universe (all IPv6 addresses).

Putting the above in practical terms, the filter of a multicast router will operate in either of two modes, *INCLUDE* or *EXCLUDE*. When in *INCLUDE* mode, the filter will allow the sources listed while blocking the rest; and when in *EXCLUDE* mode, it will block the sources listed while allowing the rest. These modes are mutually exclusive because each of them can unambiguously map the addresses listed in the filter to the master set our

[6] The complement of a set is the difference between the universe and that set.

theory is based on here: For *INCLUDE* the mapping is identity and for *EXCLUDE* it is the complement.

> Generally speaking, these two modes may have to be combined in one filter only if dealing with variable-size address ranges such as prefixes. One often encounters that case when setting up network security devices such as firewalls. It doesn't apply to SFM though because SFM is concerned only with individual source addresses.

A listener on its part will also be filtering the group's traffic in either of the two modes because the filter size considerations apply to it too. Then SFM support in MLD will boil down to implementing the following components: a wire protocol to convey the information about listener filters to the router and an algorithm for the router to combine individual listener reports into one router filter so that all of the listeners are happy.

> According to this filtering model, ASM listener mode, in which all sources are allowed, is just one case of SFM, as expressed by the filter *EXCLUDE*($\emptyset$) where $\emptyset$ is an empty set. To put the same idea in the human language, ASM is when all sources are allowed without exception (implying that such exception still is technically possible).

To communicate the filter information from a listener to the routers, let each record in an MLDv2 Report contain, in addition to the multicast address reported, a list of source addresses. To a first approximation, this list will be just the same as the source filter applied to the given multicast address by the listener reporting. Consequently, the filtering mode will need to be indicated along with the list: *INCLUDE* or *EXCLUDE*. This will be done using the **Record Type** field as shown in Fig. 6.8.

To initially upload a source filter to the routers, all of the source addresses will have to be shown and the filter mode indicated. This function will be carried out by the record types *CHANGE_TO_INCLUDE_MODE* (Type 3) and *CHANGE_TO_EXCLUDE_MODE* (Type 4). The former tells that the listener has set its source filter for the given group to *INCLUDE* mode and loaded it with the source list shown; the latter does the same thing for a filter in *EXCLUDE* mode. For convenience, let's shorthand these two Record Types as *TO_IN*(S) and *TO_EX*(S), respectively, where S is the source address set listed in the record.

Although those two record types would be sufficient, reloading the source list from scratch each time would be far from optimal. So let's provide for two more cases of filter communication.

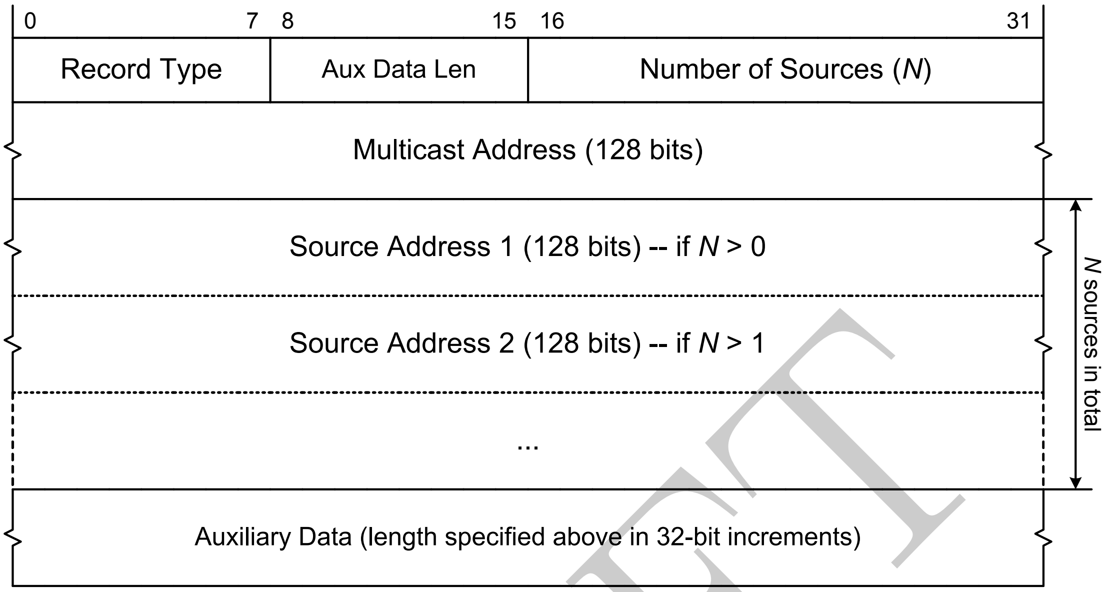

0
7
8
15
16
31
Record Type
Aux Data Len
Number of Sources (N)
Multicast Address (128 bits)
Source Address 1 (128 bits) -- if N > 0
Source Address 2 (128 bits) -- if N > 1
...
N sources in total
Auxiliary Data (length specified above in 32-bit increments)

Figure 6.8: Multicast Address Record format

One case is when a listener wants to allow or block a few more sources while keeping the rest of the filter entries intact. Such an incremental change will be communicated using the record types *ALLOW_NEW_SOURCES* (Type 5) and *BLOCK_OLD_SOURCES* (Type 6), to be shorthanded as $ALLOW(S)$ and $BLOCK(S)$, respectively.

The other case to optimize is responding to a Query, when the current filter already loaded and running just needs to be reported. This deserves a dedicated pair of record types, *MODE_IS_INCLUDE* (Type 1) and *MODE_IS_EXCLUDE* (Type 2); shorthanded as $IS\_IN(S)$ and $IS\_EX(S)$, respectively. The rationale is as follows. First, the routers will have a chance to optimize Report processing as long as the listener filters remain unchanged. Second, these two record types must not trigger further Queries because they might lead to an infinite loop.

To avoid having to repeat, "a multicast address record", all the time as well as confusing it with the record about a multicast group in a node's memory, let's just be saying, "a report", as in, "an *IS_EX* report", while keeping in mind that an MLDv2 Report message is built from one or more simple reports, each about one multicast group.

With so many report types available, a listener can combine reports about the same multicast group to signal a change in its source filter to the routers efficiently, using as few MLDv2 messages as possible. For example, when the filter is changed from $INCLUDE(A)$ to $INCLUDE(B)$, where A and B can

Table 6.3: MLDv2 listener reporting rules

Current state	New state	Report(s)
Non-listener or $\mathit{EXCLUDE}(A)$	$\mathit{INCLUDE}(B)$	$\mathit{TO\_IN}(B)$
Non-listener or $\mathit{INCLUDE}(A)$	$\mathit{EXCLUDE}(B)$	$\mathit{TO\_EX}(B)$
$\mathit{INCLUDE}(A)$	$\mathit{INCLUDE}(B)$	$\mathit{BLOCK}(A \setminus B)$, $\mathit{ALLOW}(B \setminus A)$
$\mathit{EXCLUDE}(A)$	$\mathit{EXCLUDE}(B)$	$\mathit{ALLOW}(A \setminus B)$, $\mathit{BLOCK}(B \setminus A)$
$\mathit{INCLUDE}(A)$ or $\mathit{EXCLUDE}(A)$	Non-listener	$\mathit{TO\_IN}(\emptyset)$

be somewhat different sets of source addresses, there will be two subsets of sources affected. Set $A \setminus B$ (A except any elements present in B) is no longer interesting to the listener and can be blocked by the routers unless there are more listeners interested in those sources. On the other hand, set $B \setminus A$ (B except any elements present in A) wasn't interesting to the listener before, and so the routers will need to ensure all of its elements are unblocked. At the same time, the intersection $A \cap B$ (all elements common to A and B) remains as interesting to the listener as before, requiring no action from the routers. The change in question thus can be signaled to the routers in an incremental manner using two reports sent in any order, $\mathit{BLOCK}(A \setminus B)$ and $\mathit{ALLOW}(B \setminus A)$. These reports can be wrapped in one MLDv2 Report message, the packet length permitting.

Visualize this example with an Euler diagram.

The same change can be communicated with just one report, $\mathit{TO\_IN}(B)$, but this report implies that the filter mode has changed as well, and the routers will start additional inquiries. Furthermore, if the filter is long but only a few entries have changed in it, the total length of the *ALLOW* and *BLOCK* can be considerably less than that of the *TO_IN* because the former pair of reports can communicate incremental changes instead of always reloading the filter from scratch. For these reasons, a listener shouldn't be using *TO_IN* or *TO_EX* to alter an existing source filter.

Now that we have got hold of the idea, let's compile a complete system of reporting rules for an MLDv2-capable listener in Table 6.3. As usual, $A \setminus B$ will stand for a set difference, i.e., all elements of A not found in B; and $\emptyset$ will denote an empty set.

In practice, there can be extra complexity for the listener to handle stemming from the requirement to repeat a report a few times in case some of its messages are lost by the link. Should the group change its state while being reported, the new report will need to be merged with the old one based on the same set theory considerations [191, Section 6.1].

SSM is a bit special when it comes to reporting the filter. Namely, there should be no *IS_EX* or *TO_EX* reports about the *FF3x::/96* multicast addresses. The point is that an SSM listener is supposed to be interested only in a limited set of sources, and so its source filter is restricted to *INCLUDE* mode [198, Section 2.2.2].

And now we are to face the most challenging problem posed by the MLDv2 design: how a router should merge reports about the same group from different listeners so that all of them are happy. To probe this problem, let's first consider a few particular cases yielding to trivial analysis.

If all of the listeners are in the same mode with respect to the given multicast group, be it *INCLUDE* or *EXCLUDE*, the answer can be provided by set theory alone. As we have already said, the router must allow the union of the listeners' master sets (the theoretical sets of source addresses allowed by each listener) to make sure no authorized source ever gets blocked, and precise filtering to match each listener's needs will be conducted by the listener itself. When all listeners are in *INCLUDE* mode, the merge operation to yield the router filter R, also in *INCLUDE* mode, is just the union of their *INCLUDE* filters I_i:

$$R = \bigcup_i I_i.$$

On the other hand, when all listeners are in *EXCLUDE* mode, the best the router can do is *block* the intersection of their *EXCLUDE* filters E_i:

$$\overline{R} = \bigcap_i E_i.$$

This conclusion is based on the following corollary of De Morgan's laws:

$$(A \setminus B) \cup (A \setminus C) = A \setminus (B \cap C).$$

If A is the set of all IPv6 addresses while B and C are two *EXCLUDE* filters regarded as sets, the meaning of this equation can be explained as follows: the intersection of *EXCLUDE* filters is equivalent to the union of the corresponding *INCLUDE* filters, which are the same as the master sets.

But what if the listeners of the group aren't all in the same mode? The rule of allowing the union of their master sets is fundamental and can be applied to this case too. Quite obviously, a union of sets can't have fewer elements in it than the largest of the original sets had. The master set for any *EXCLUDE* filter of a practical length will include almost all IPv6 addresses, and so will the resulting union to allow. As we already know, a practical way to handle such an enormous filter is blocking the remaining addresses instead. Consequently, the suitable mode for the router filter will be *EXCLUDE* in this case. In other words, *as soon as there is a listener in EXCLUDE mode on the link, the routers have to switch to EXCLUDE mode as well.* Based on the above results, the list of source addresses to block so can be computed as the intersection of all *EXCLUDE* filters less the union of all *INCLUDE* filters:

$$\overline{R} = \bigcap_i E_i \setminus \bigcup_i I_i.$$

In particular, as long as there are ASM listeners on the link ($\exists n : E_n = \emptyset$), the resulting filter will be just empty, preventing the routers from blocking any source. On the other hand, when the last listener in *EXCLUDE* mode is gone, the routers should switch back to *INCLUDE* mode for finer filtering. Should the listeners in *INCLUDE* mode leave eventually as well, the *INCLUDE* filter as computed by the routers will turn empty ($R = \emptyset$), which means the end of forwarding the given group to the link.

This is it about the set-theory background of SSM/SFM support by MLDv2, the rest being various practical considerations. Among them, this one really stands out: how a router would manage the information about multicast groups to keep it up to date and suitable for all listeners in the face of various events such as listeners joining in, leaving, crashing, reporting changes, etc, etc, etc. Back in MLDv1, it was sufficient to have just one timer ticking on each group and, when a listener was leaving a group, to verify with a group-specific Query if there were any more listeners left in that group. So the basic management object of MLDv1 was a group. The MLDv2 reality is that different listeners request different sources be allowed or blocked for each group, but the router still won't track each listener's preferences to be able to handle a potentially large number of listeners with as little resources as possible. Therefore a separate timer will be needed on each source learned for a group [191, Section 7.2], and a source sending to a specific group will become the basic management object of MLDv2, as illustrated in Fig. 6.9.

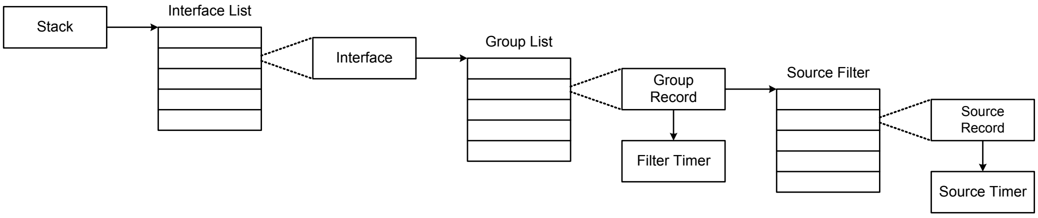


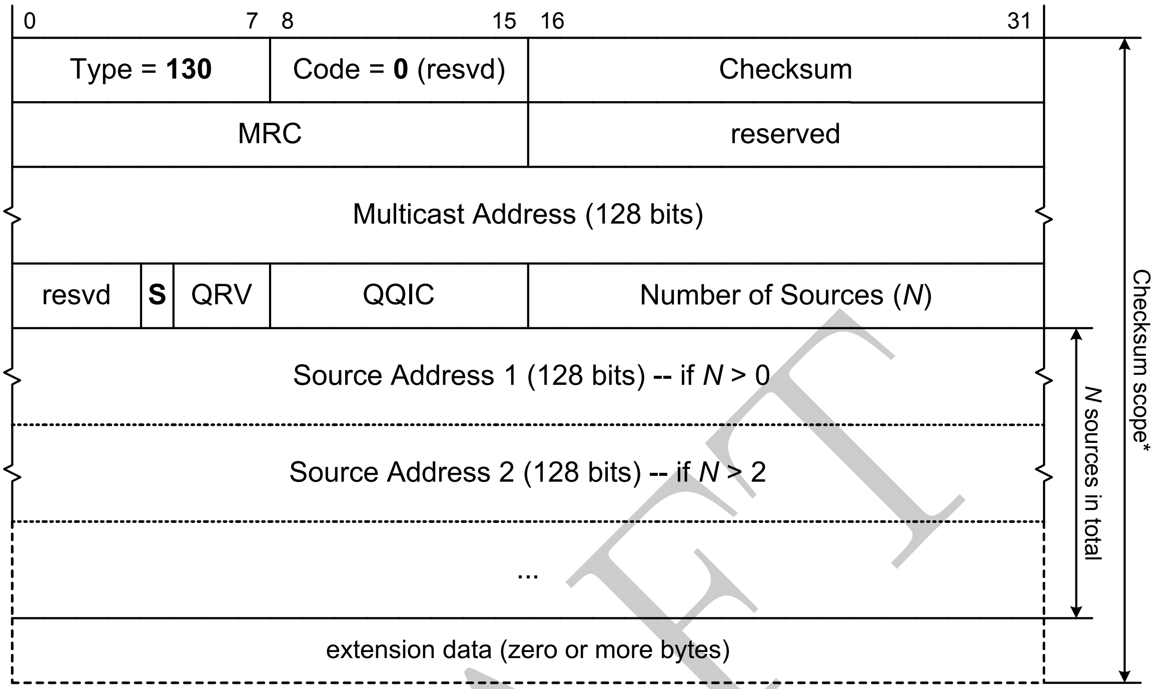
Figure 6.9: Conceptual data structures of an MLDv2-capable router

> Note well that different multicast groups have completely independent source filters, and so the same source can be allowed to send to one group while being blocked by another. The same IP address thus can appear in different source records in router's memory if they are linked with different groups.

When any of the listeners sends through an *IS_IN*(A), *TO_IN*(A), or *ALLOW*(A) report to indicate that it wants to be receiving packets from a set of sources, A, the router can do nothing but unblock those sources immediately. By contrast, when a listener sends through an *IS_EX*(B), *TO_EX*(B), or *BLOCK*(B) report to tell the router it isn't interested in traffic from a set of sources, B, the router can't block those sources right away because there may be other listeners still interested in at least some sources from set B. The idea is basically the same as in MLDv1, where a router couldn't just cease forwarding packets to a group upon reception of a Done message: it had to confirm first there were no other listeners. This has to do with the multicast router's refusal to track individual listeners.

Nevertheless, an *IS_EX*(B), *TO_EX*(B), or *BLOCK*(B) report doesn't really oblige the router to check each of the sources from set B because the router may already be handling some of those sources right due to earlier reports from other listeners. The router has already got a certain master set $\mathbb{S}$ consisting of all sources to be allowed, and encoded in *INCLUDE* or *EXCLUDE* mode. (As we said, in *INCLUDE* mode the master set is the same as the filter list: $\mathbb{S} = R$; and in *EXCLUDE* mode it is the filter's complement in the universe of this problem, i.e., in the IPv6 address space: $\mathbb{S} = \backslash\overline{R}$.) Therefore a check is required only for the intersection of B and $\mathbb{S}$, $Q = B \cap \mathbb{S}$, because the sources outside this set Q aren't going to be affected by the change anyway: subset $\mathbb{S} \setminus B$ remains allowed while subset $B \setminus \mathbb{S}$ has already been blocked.

To conduct a selective check on a set of sources, the router will need to send a new form of Query that can be limited not only to a specific group


0
7
8
15
16
31
Type = **130**
Code = **0** (resvd)
Checksum
MRC
reserved
Multicast Address (128 bits)
resvd
S
QRV
QQIC
Number of Sources (*N*)
Source Address 1 (128 bits) -- if *N* > 0
Source Address 2 (128 bits) -- if *N* > 2
...
extension data (zero or more bytes)
N sources in total
Checksum scope*

*) Keep in mind that Checksum also protects the pseudo-header.

Figure 6.10: MLDv2 Query format

but also to a list of sources. It appears for the first time in MLDv2 and is known as **Multicast Address and Source Specific Query**. Will such a Query need to have a filtering mode, *INCLUDE* or *EXCLUDE*, specified? In fact, the router is only interested in hearing back about allowed sources because blocking is the default—this follows from the point of MLD, which is to get rid of as much unwanted multicast traffic as possible. As for the set to query about, $Q = B \cap \$$, it is no larger than the set B from the report and so it can be shown as is, by just listing its elements. Consequently, a Multicast Address and Source Specific Query will always come in *INCLUDE* mode, making a mode field redundant. Its format, complete with some fields to be introduced later on, is shown in Fig. 6.10. Let such a Query limited to group G and source set Q be denoted by $Q(G, Q)$ while a Query limited only to group G will be denoted by $Q(G)$. The latter Query variant will contain zero sources, effectively an empty set of them.

Although we have just found valid, theory-backed reasons to confine $Q(G, Q)$ to *INCLUDE* mode, that turns out to be at odds with the *EXCLUDE* mode of the router filter and increases the complexity of a source

Table 6.4: Conceptual modes of an MLDv2 router filter

Mode	Parameter meaning
INCLUDE(A)	A **Include List**. This is a list of all sources allowed to send to this group. Any other sources are blocked by the router.
EXCLUDE(X, Y)	X **Requested List**. This is a list of sources that might still be interesting to some listeners. Such sources are being verified with a Query rather than just blocked. Later we will encounter another function of the Requested List. Y **Exclude List**. This is a list of all blocked sources. Any other source, including those from list X, is allowed to send to this group.

check operation. As long as the router filter is in *INCLUDE* mode, checking individual sources is straightforward: the set to check, Q, is known to be a subset of the current filter ($B \cap \$ \subseteq \$$), thus guaranteeing there is a record about each source from Q in the router's memory. Then the check can boil down to sending a suitable Query and reducing the timer interval on each source from Q. By contrast, when the router filter is in *EXCLUDE* mode, the elements of Q will never be found in the current filter ($B \cap \$ \not\subset \setminus\$$), making the router keep their records separately from the filter proper.

To hold on to all information the router may need, just one source list is enough when in *INCLUDE* mode but as many as two of them are required when in *EXCLUDE* mode. The well-known notation for those modes is shown in Table 6.4 [191, Section 2.3].

When in *EXCLUDE*(X, Y) mode, lists X and Y have no common elements: $X \cap Y = \emptyset$. It is so simply because no source can be allowed and blocked at the same time. Consequently, list X has no immediate effect on the filtering as such. However, should one of its entries, $x \in X$, age out, the router's conclusion will be that source x is no longer interesting to any listener. Then the router will be permitted to move x from list X over to list Y, thus blocking this source and relieving the link of some unwanted traffic. This is what list X is for.

Now we are almost ready to go ahead and specify the rules governing how a router should handle listener reports. Those rules will be quite elaborate because they have to take account of all our previous considerations. Still, we don't need to worry about their complexity as we will take time to elucidate each rule. Before taking the plunge we need to fix up a few auxiliary details though.

As was said, the basic management object of MLDv2 is a source sending to a specific group. A multicast router is tasked with minimizing stray traffic into the link by blocking unwanted sources for each group. The router dreams of blocking all traffic, if you will, but is prevented from doing so by listener reports requesting this or that source allowed.

As long as a source has a record in the Include List or the Requested List for a given group, the router is explicitly obliged to let that source to the group, patiently waiting to block it as soon as it is no longer listened to. To ensure this, a timer is started on the record, to fire if there are no reports about this source before the interval elapses. What happens to the expired record will depend on the filter mode, but in any event the source will get blocked. Namely, in *INCLUDE* mode the record will be just dropped from the Include List whereas in *EXCLUDE* mode it will be moved over to the Exclude List.

> Let's reiterate that a source record is bound to a specific group. For example, the same source S can be allowed for group G_1 and blocked for group G_2. In the IPv6 header terms, packets from S to G_1 will be forwarded by the router into the link, but packets from S to G_2 will not. This is reflected in how the conceptual structures from Fig. 6.9 are organized in the router memory.

While in the Exclude List, a source record causes the source blocked and the traffic into the link reduced somewhat. This is what the router is after, so this state shouldn't be timed out. If a listener ever needs that source again, it will have to send a report through. This is at the essence of the MLD model: listeners have to keep bothering the router as long as they need the traffic because the default for the router is to block it. Consequently, the timers on Exclude List entries are just stopped.

At the same time, the Exclude List comes along with *EXCLUDE* mode, which is as far from the router's sweet dream of blocking all sources as the sky. Indeed, only the N sources currently in the Exclude List may be blocked while the rest, up to $2^{128} - N$ of them, are free to send packets to the group. What a nightmare for the poor router! So it will be in its best interests to switch the filter to *INCLUDE* mode as soon as the last listener in *EXCLUDE* mode is gone. Tracking no individual listeners, the router will have to rely on yet another timer here. It is a so-called **Filter Timer** linked to the group record, as was shown in Fig. 6.9. It will be ticking only while in *EXCLUDE* mode and get reset upon reception of an *IS_EX* or *TO_EX* report because such a report indicates there still are *EXCLUDE*-mode listeners of the group on the link. On the other hand, should the Filter Timer interval expire, the router will get full right to switch to *INCLUDE* mode itself for the given group.

Notwithstanding its eagerness to block all traffic to the group, the router must allow sources on request by listeners. In particular, it is supposed to block no source recently seen in an *IS_IN*, *TO_IN*, or *ALLOW* report. Therefore, when flipping the filter mode to *INCLUDE*, the router should have populated the Include List with all of the authorized sources; otherwise they will immediately get blocked by default. As we see now, the filter has to collect this kind of information well in advance, while still in *EXCLUDE* mode. Will it need an extra list for that? Let's try to make do with the Requested List instead in the hope that it can play both roles. There are no fundamental obstacles to the router's keeping authorized sources in that list. However, they can get mixed up, when in there, with the sources the router is trying to get blocked. To avoid as much of the confusion as possible, let's make use of the following observation: If an *IS_EX* or *TO_EX* report has just come through, the filter will remain in *EXCLUDE* mode at least for another Filter Timer interval. So, if this interval is made the same as the source record timeout, the *INCLUDE*-mode listeners will have plenty of time to re-report their preferences in response to a periodic General Query. That being the case, the router can safely drop all entries from the Requested List when an *IS_EX* or *TO_EX* report is received, except for the current to-block candidates from the latest report [191, Section 7.2.3].

We have taken the liberty of speaking about MLD timing in common-sense terms such as "recently" or "long ago". In most cases, it is clear that "recently" means, "while the timer interval hasn't expired", which timer clear from the context. Still, it is time we came up with more precise details as to how MLDv2 is to be timed. As we have agreed, each group tracked by the router will need two kinds of timers.

One of them is per source record. When in *INCLUDE* mode, it will play about the same role as the group timer in MLDv1, namely, to let the router detect when it is safe to block the traffic because no listener is interested in it. The difference between the two versions of MLD is just that MLDv1 would manage whole groups while its successor can do a finer job by managing individual sources sending to each group. So, when such a timer goes off, MLDv1 would prune the entire group while MLDv2 will just drop a single source. When the *INCLUDE*(A) filter ultimately becomes empty due to all of its entries having aged out, it will be time to remove the group record altogether, so a separate group timer would be redundant in MLDv2.

How long will the interval of a source timer need to be? Such a timer gets reset when a report is received telling that this source is interesting to some listener of this group, so its interval should be longer than the period between such reports; otherwise the filter won't stabilize, flapping instead. As long as no listeners change their state, their reports are triggered by a periodic General Query by the current Querier router. This is a quite expensive operation, with each listener reporting all its groups and associated source filters using the *IS_IN* or *IS_EX* report types, and so the **Query Interval** parameter, **QI** for short, should be quite conservative, defaulting to 125 seconds [191, Section 9.2].

An additional consideration is that a single Query may not be received by all listeners because of a link-level fault or congestion, and a Report message may as well be lost. Still, there is hope that one of the subsequent Queries will make it through and trigger the missing reports. So the router should wait for several QI intervals before declaring the source unwanted and dropping its record. The number of attempts required so is specified by the **Robustness Variable** parameter (**RV**), which depends on how unreliable the link is and defaults to 2 [191, Section 9.2]. Thus up to $RV - 1$ losses in a row can be tolerated.

Lastly, the listeners shouldn't respond to a Query all together, instead delaying their reports for a random time to smooth a potential traffic burst out. The upper bound for such a random delay is given by the **Query Response Interval** parameter (**QRI**), which defaults to 10 seconds [191, Section 9.3].

Taking all these factors into account, the source timer interval shouldn't be less than $QI \times RV + QRI$ as long as the network is stable; using the default values, it comes to 260 seconds. This is known as the **Multicast Address Listening Interval**, or **MALI** for short [191, Section 9.4].

> The QI interval has to appear in $MALI$ exactly RV times, not $RV - 1$. This is because the creation of the source record may not be aligned in time with the Query sending period. So the next scheduled Query will be sent no later than in QI seconds, followed by $RV - 1$ intervals between its retransmits. Hence RV intervals of QI in total.

> What factor is overlooked by the $MALI$ formula? (Hint: *The link transmission delay.*)

Suppose now that the router receives a report casting doubt upon some source records, e.g., a *BLOCK* report. The router will respond to that with active checks on those records using a Multicast Address and Source Specific Query. The MLDv1 equivalent of this scenario was triggered by reception of a Done message. In order to get a reliable result in the shortest time possible, the router will send the Query message **LLQC** times (**Last Listener Query Count**, by default equals to RV [191, Section 9.9]), but making pauses between the Queries as short as **LLQI** (**Last Listener Query Interval**), just 1 second by default [191, Section 9.8]. The listeners on their part will delay their responses to such a Query for a short random time between 0 and $LLQI$, assuming that a request for a shorter delay interval can be signaled to them in the Query in about the same manner as in MLDv1. In this scheme, the longest it can take a Report message to come through is $LLQI \times LLQC$ since the start of the check, i.e., 3 seconds by default. This interval is denoted by **LLQT** (**Last Listener Query Time**) [191, Section 9.10]. There will be little sense in using a source timer interval longer than $LLQT$ during a fast check phase.

> We can't resist suggesting that $LLQT$ be pronounced, "Li'l cutie".

> The $LLQI$ interval is included in $LLQT$ exactly $LLQC$ times, not $LLQC + 1$, because the first Query will be sent immediately. This will be followed by $LLQC - 1$ intervals between Queries, followed in turn by another interval to allow for a random delay in listeners reporting back.

So it turns out that, depending on the situation, the optimal source timer interval can be long ($MALI$) or short ($LLQT$). Until we are concerned with router synchronization, the procedure of selecting the appropriate interval can be as simple as follows:

- By default, a source timer is set to the long interval ($MALI$).
- When the first message in a series of $Q(G, S)$ Queries specific to group G and source set S is sent, the timer is reset to the short interval ($LLQT$) on each source from set S for group G unless its remaining interval is already $LLQT$ or less.

> We will have to alter this nice procedure somewhat to provide for reliable synchronization between the Querier and non-queriers. In some cases, the source timer will need to be reset to *LLQT* after a few subsequent Queries in a series in addition to after the opening one.

This way the router's sending a Multicast Address and Source Specific Query serves as a direct indication that there is doubt whether the sources listed in the Query still are interesting to any listener of the group.

Great, now the router can exercise precise control over the source timers while in *INCLUDE* mode. Will the rules for *EXCLUDE* mode have to be completely different? Let's first recall that in the latter mode instead of just one Include List there are as many as two source lists for each group, the Requested List and the Exclude List, each source record to be found in either of them. The role of the Requested List is such that it is to be managed in almost the same way as the Include List. The only significant difference is that on timeout a Requested List entry is moved over to the Exclude List rather than just dropped because it has been a candidate for blocking. Once in the Exclude List, a source record becomes, if you will, immortal, with its timer not running.

Nonetheless, there will be a timer ticking when in *EXCLUDE* mode. It is the other kind of MLDv2 timer we suggested above, namely, the Filter Timer ensuring that the filter won't linger in *EXCLUDE* mode after the last *EXCLUDE*-mode listener of the group has disappeared. As we have found out, for the sake of protocol stability the Filter Timer interval needs to be the same as a source timer interval; but which of them, the longer or the shorter?

As long as the configuration of the listeners remains constant, it should be the long interval ($MALI$) because a prerequisite to a safe $EXCLUDE \rightarrow INCLUDE$ transition is that the Requested List have been populated with authorized sources, and for that each *INCLUDE*-mode listener needs to get a chance to send an *IS_IN* report in response to the most recent of periodic Queries.

Still, a situation is possible where the router gets a reason for doubt that it has to remain in *EXCLUDE* mode any longer. Such a reason comes in the shape of a *TO_IN* report telling that the number of *EXCLUDE*-mode listeners might have reduced. The natural reaction by the router will be to verify if their number finally dropped to zero. To perform this check, the router will need to send a series of Multicast Address Specific Queries, $Q(G)$, intending to trigger *IS_EX* reports from the remaining *EXCLUDE*-mode listeners, if any, within $LLQT$. Such Queries will be responded to by the *INCLUDE*-mode listeners of group G as well, making it safe to reduce the $EXCLUDE \to INCLUDE$ transition time to just $LLQT$ in case no *IS_EX* reports are heard back.

Let's sum up these considerations in the procedure to control a Filter Timer interval with—it will turn out to be nearly the same as that for the source timer interval:

- By default, a Filter Timer is set to the long interval ($MALI$).
- When the first message in a series of $Q(G)$ Queries specific only to group G is sent, the Filter Timer of group G is reset to the short interval ($LLQT$) unless its remaining interval is already $LLQT$ or less.

> We will have to alter this procedure, too, to provide for reliable synchronization between the Querier and non-queriers. In some cases, the Filter Timer will need to be reset to $LLQT$ after a few subsequent Queries in a series in addition to after the opening one.

Quite remarkably, neither timer management procedure really cares about the current filter mode because a filter in *EXCLUDE* mode will never need a $Q(G, S)$ Query while a filter in *INCLUDE* mode will never cause a $Q(G)$ Query.

> Still, extra checks won't hurt in practice, especially if different modules of the implementation were created by different developer teams.

So the MLDv2 Query variants end up each having a role of its own, as shown in Table 6.5.

Now we are finally ready to formulate the system of report processing rules for an MLDv2-capable router as shown in Table 6.6.

> A task for the reader will be to draw and analyze the Euler diagrams corresponding to the rules from Table 6.6.

Table 6.5: MLDv2 Query variants

Query variant	Symbol	When to send	Role
General Query	$Q(::)$	At router startup, then periodically	To learn the current state of things on the link
Multicast Address Specific Query	$Q(G)$	When the number of listeners of group G in *EXCLUDE* mode might have reduced	To check if there are any listeners left in *EXCLUDE* mode
Multicast Address and Source Specific Query	$Q(G, S)$	When the number of listeners of group G interested in sources from set S might have reduced	To find out whether those sources are still interesting to any listener of that group

Table 6.6: MLDv2 report processing rules

<table>
<tr><th>Current state</th><th>Report</th><th>New state</th><th>Queries</th><th>Delete</th><th>Timers</th></tr>
<tr><td>$INCLUDE(A)$</td><td>$ALLOW(B)$</td><td>$INCLUDE(A \cup B)$</td><td>—</td><td>—</td><td>$(B) = MALI$</td></tr>
<tr><td>$INCLUDE(A)$</td><td>$IS\_IN(B)$</td><td>$INCLUDE(A \cup B)$</td><td>—</td><td>—</td><td>$(B) = MALI$</td></tr>
<tr><td>$INCLUDE(A)$</td><td>$TO\_IN(B)$</td><td>$INCLUDE(A \cup B)$</td><td>$Q(G, A \setminus B)$</td><td>—</td><td>$(B) = MALI$</td></tr>
<tr><td colspan="6"><b>Comment:</b> The current filter accepts only sources from set A. All three Reports essentially request the immediate extending of the allowed set to $A \cup B$. The timers are reset on the source records new information has been received about. The $TO\_IN(B)$ Report also triggers a Query about the rest of the allowed sources, $Q(G, A \setminus B)$, because such a Report may be a signal that the set of sources listened to has reduced rather than extended. On the other hand, the $IS\_IN(B)$ Report triggers no such Query to avoid a protocol loop since that Report itself was likely to be prompted by a Query.</td></tr>
<tr><td>$INCLUDE(A)$</td><td>$BLOCK(B)$</td><td>$INCLUDE(A)$</td><td>$Q(G, A \cap B)$</td><td>—</td><td>—</td></tr>
<tr><td colspan="6"><b>Comment:</b> The current filter accepting only sources from set A, subset $B \setminus A$ is already blocked and so it is enough to consider just subset $A \cap B$. However, this subset can't be blocked immediately because other listeners may still be interested in sources from it. For this reason only a Query $Q(G, A \cap B)$ is sent while the filter remains unchanged.</td></tr>
</table>

Table 6.6: MLDv2 report processing rules (continued)

Current state	Report	New state	Queries	Delete	Timers
INCLUDE(A)	*IS_EX*(B)	*EXCLUDE*$(A \cap B, B \setminus A)$	—	$A \setminus B$	$(B \setminus A) = 0$, $(Filter) = MALI$
INCLUDE(A)	*TO_EX*(B)	*EXCLUDE*$(A \cap B, B \setminus A)$	$Q(G, A \cap B)$	$A \setminus B$	$(B \setminus A) = 0$, $(Filter) = MALI$

Comment: First of all, the filter has to be switched to *EXCLUDE* mode because a listener in this mode has been detected. All sources but those from set A have already been blocked by the current filter, and so it is safe for the new filter to block the following subset of B: $B \cap (\setminus A) = B \setminus A$. The timers on these source records get stopped because the Exclude List has just one timer, which is reset and started. The rest of B, i.e., $A \cap B$, can't be just blocked, but it can be verified instead; so it is saved in the Requested List. Still, an actual Query about it may be sent only if the Report was about a listener filter change (*TO_EX*) and not its current state (*IS_EX*) because the latter was triggered by another Query. With the new filter installed, the sources from subset $A \setminus B$ will be allowed by default and so their individual records can be deleted.

Table 6.6: MLDv2 report processing rules (continued)

Current state	Report	New state	Queries	Delete	Timers
EXCLUDE(X, Y)	*ALLOW*(A)	*EXCLUDE*$(X \cup A, Y \setminus A)$	—	—	$(A) = MALI$
EXCLUDE(X, Y)	*IS_IN*(A)	*EXCLUDE*$(X \cup A, Y \setminus A)$	—	—	$(A) = MALI$
EXCLUDE(X, Y)	*TO_IN*(A)	*EXCLUDE*$(X \cup A, Y \setminus A)$	$Q(G, X \setminus A)$, $Q(G)$	—	$(A) = MALI$

Comment: Here we meet the other role of the Requested List X, which is to accumulate allowed source records in case the filter switches to *INCLUDE* mode. Therefore such records are entered into the Requested List with a set union operation: $X \cup A$. The timers are reset on all records up-to-date information has been learned about. At the same time any elements of set A are removed from the Exclude List Y, which is reduced thus to $Y \setminus A$. Indeed, the Report received demands set A unblocked, which can't be delayed because the router filter may err only on the permissive side.

The *IS_IN* Report triggers no Query as it was prompted by a Query itself. By contrast, the *TO_IN* Report leads to a Query about the elements of X no fresh information is available about: X except A. Such a Query is mostly needed to prevent bloating the Requested List with outdated entries. In addition, the *TO_IN* Report can indicate that the number of listeners in *EXCLUDE* mode has reduced if one of them switched to *INCLUDE* mode. To verify whether any *EXCLUDE*-mode listeners of G are still left out there, a Multicast Address Specific Query $Q(G)$ is sent as well. Although it may seem to make the Multicast Address and Source Specific Query $Q(G, X \setminus A)$ redundant, the latter is still needed for timer synchronization between the routers of the link—a topic to be discussed soon.

Table 6.6: MLDv2 report processing rules (continued)

Current state	Report	New state	Queries	Delete	Timers
EXCLUDE(X, Y)	*IS_EX*(A)	*EXCLUDE*$(A \setminus Y, Y \cap A)$	—	$X \setminus A$, $Y \setminus A$	$(A \setminus X \setminus Y) = MALI$, $(Filter) = MALI$
EXCLUDE(X, Y)	*TO_EX*(A)	*EXCLUDE*$(A \setminus Y, Y \cap A)$	$Q(G, A \setminus Y)$	$X \setminus A$, $Y \setminus A$	$(A \setminus X \setminus Y) = (Filter)$, $(Filter) = MALI$

Comment: It is only safe to keep blocking the intersection $Y \cap A$ as all listeners agree only about it. The rest of set Y, that is, $Y \setminus A$, must be unblocked without delay because the reporting listener is expressly interested in it. The rest of set A, that is, $A \setminus Y$, should be verified with a Query first, providing the Report is about a listener filter change, not about its current state.

The role of the Requested List is flipped again: It will keep track of the sources being verified while all other entries get dropped from it. In particular, subset $X \setminus A$ is removed from the Requested List because those sources can't be blocked now that the listener has expressed its interest in them. At the same time, subset $X \cap A$ stays in the Requested List because it is a candidate for blocking.

The timers are started on the source records newly appearing in the Requested List, $A \setminus X \setminus Y$. In the case of the *TO_EX* Report their intervals are set to the *current* value of the Filter Timer. Why so? As long as the router filter is in *EXCLUDE* mode, its Filter Timer is counting down the time left until the sweet moment when the router gets right to block nearly all sources. In essence, the Filter Timer is ticking on each source that isn't listed in X or Y and so is allowed by default. To preserve this valuable information, the remaining Filter Timer interval needs to be copied to the source's own timer when its individual record is created in the Requested List. This is what happens here: just the preservation of the information accumulated.

Table 6.6: MLDv2 report processing rules (continued)

Current state	Report	New state	Queries	Delete	Timers

Comment (continued): The *IS_EX* case is special. If such a message comes through, reporting the listener's current state rather than a change of it, and subset $A \setminus X \setminus Y$ is found not empty, then the router is out of sync with the listener. What might have happened is that the router has started up only recently or there has been a link fault; but the most probable cause of the out-of-sync condition is that the router deliberately discarded the information from the Requested List when switching its role to source verification. For example, this will be the case when there are multiple *EXCLUDE* mode listeners on the link and their filters aren't the same—the Requested List will fluctuate, never stabilizing at one set of sources. (To see how it would happen, study the scenario of two listeners where one listener blocks set $\{1, 2\}$ while the other listener blocks set $\{2, 3\}$.) In any event, the router can't fully trust the data it currently has, and so the timers on the new records are just set to the conservative long interval, *MALI*. (Still, we conjecture that the timer setting $(A \setminus X \setminus Y) = (Filter)$ would cause no problem in this case either. Explore this conjecture, time permitting.) Besides, no Query is sent because this Report was caused by a Query.

Finally, either Report proves there is at least one listener in *EXCLUDE* mode and so the Filter Timer gets reset.

Table 6.6: MLDv2 report processing rules (continued)

Current state	Report	New state	Queries	Delete	Timers
$EXCLUDE(X, Y)$	$BLOCK(A)$	$EXCLUDE(X \cup (A \setminus Y), Y)$	$Q(G, A \setminus Y)$	—	$(A \setminus X \setminus Y) = (Filter)$

Comment: This rule is similar to the previous one, *TO_EX*. A part of set A may already be blocked by the Exclude List Y, so the changes concern subset $A \setminus Y$ only. As usual, instead of the immediate blocking of the subset, a Query is sent about it. However, a *BLOCK* Report can be coming from a listener in either mode, *INCLUDE* or *EXCLUDE*; so the router just adds the missing entries to the Requested List using a set union operation rather than resets it to $A \setminus Y$ each time. It is for the same reason that the Filter Timer isn't reset, ticking on instead; a *BLOCK* Report does nothing to indicate there are listeners in *EXCLUDE* mode. Should the Filter Timer expire soon, the router filter will switch to *INCLUDE* mode and the Include List will be populated from the former Requested List with all of the allowed sources recently learned. This way the router will ensure no allowed sources get blocked when the filter has to switch its mode. As for the timer setting rationale, see the above discussion of *EXCLUDE*/*TO_EX*.

Unlike in the *TO_EX* rule, the Exclude List requires no immediate change here. The *TO_EX* rule required it because the *TO_EX*(A) Report also tells the router that this listener is interested in all sources but from set A and the router must unblock them forthwith. On the other hand, the *BLOCK*(A) Report conveys no such request to the router, so that no sources need to be unblocked when processing this Report.

Another problem for the reader is to confirm that MLDv2 Reports are idempotent, that is, multiple copies of the same Report received back to back will result in the same router filter state as just a single instance of the Report would. This property allows a listener to repeat its Report without having to wait for the next Query to come, to make sure it is received by the router in the face of a potentially lossy link.

Do you think MLDv2 Reports are commutative and/or transitive?

All Reports but *IS_EX* and *TO_EX* are additive: If the source list is split into shorter sub-lists to be sent in separate messages, the resulting state of the router filter will be the same as that from just one message with the whole list. It is clear what this property can be good for: Should an MLD Report message need to list so many sources that it ends up exceeding the link MTU, it will be safe just to split it up into several smaller messages of the same type, each listing just a subset of the original set of sources.

Still, this trick won't work for an *IS_EX* or *TO_EX* Report. What can be done about it then? It will be acceptable to send just a part of the source list in one message not exceeding the MTU [191, Section 5.2.15]. Indeed, it is OK for a router to allow more sources to send to a multicast group than requested by its listeners, and omitting some of the sources from an *IS_EX* or *TO_EX* Report is equivalent to allowing those sources.

Calculate the maximum number of source records to fit into an MLDv2 Query or Report message if the MTU is the smallest possible, 1280 bytes. (Hint: *Don't forget to allow for a Router Alert option. The answer is:* $(1280 - 40 - 8 - 28) div 16 = 75$*, where 1280 is the MTU, 40 is the IPv6 header, 8 is a Hop-by-Hop Options header with a Router Alert option, and 28 is the MLDv2 overhead—the same size for a Query and a Report.*)

As it turns out, the complexity of the MLDv2 Report processing rules (as well as of their IGMPv3 counterparts) has to do with the requirement to support listeners in $EXCLUDE(A)$ mode where set A isn't empty. As an advanced exercise, design a simplified protocol, to be known as **LW-MLDv2** (**Lightweight MLDv2**), in which a listener can either allow certain sources ($INCLUDE(A)$) or just accept all sources ($EXCLUDE(\emptyset)$) but can't block select sources. In this case the router rules will be much simplified because maintaining the Exclude List is no longer necessary. Quite surprisingly, an LW-MLDv2 capable router can interoperate with full-fledged MLDv2 listeners because the router is permitted to allow excess sources beyond what was requested by the listeners. The LW-MLDv2 router will just handle any listener Report of the form $IS\_EX(A)$ or $TO\_EX(A)$ as a request to allow all sources, that is, $IS\_EX(\emptyset)$ or $TO\_EX(\emptyset)$, respectively. Should you have difficulty "designing" LW-MLDv2 yourself, refer to [199] for its specification and discussion.

This set theory-based specification of the MLDv2 protocol has an interesting singularity. When a rule from Table 6.6 requires the router to send a Multicast Address and Source Specific Query, the source list can turn out just empty. For instance, in the last rule, $EXCLUDE/BLOCK$, it will be the case if A is a subset of Y: $A \subseteq Y \Rightarrow A \setminus Y = \emptyset$. If handled naively, this would result in the following Query: $Q(G, \emptyset)$. However, that Query would be indistinguishable from just a Multicast Address Specific Query, $Q(G)$. How can this ambiguity be resolved? As a matter of fact, the singular Query $Q(G, \emptyset)$ need not be sent in the first place because it is essentially a no-op that will never elicit a response. As we have discussed in this section, a Query is always stated in $INCLUDE$ mode and so an empty list of sources can trigger no Report by design. In other words, the only listener mode that would ever be able to match the $Q(G, \emptyset)$ Query is $INCLUDE(\emptyset)$; but it is a synonym for not listening to this multicast group at all.

If put in the now popular newspeak of Nineteen Eighty-Four, it is a non-mode of a non-listener.

Now there are just a few smaller details for us to fix up before our work on MLDv2 can be called done. So let's consult our list of proposed improvements again and see which of its items are yet to be addressed. The only such item left is the coordination of different multicast routers residing on the same link. To start cooperating, the routers need, first of all, to elect the Querier. We already have an elegant procedure for that, which considers just the routers' IP addresses. It will do the job in MLDv2 as well as it did in MLDv1.

To support this procedure, a new timer will be needed by a non-querier to be able to detect that the current Querier is gone. This timer is reset to the interval **OQPT** (**Other Querier Present Timeout**) each time a Query is received. If the timer expires, it is time to conduct an election again. The standard $OQPT$ value is $QI \times RV + \frac{1}{2}QRI$ [191, Section 9.5]. However, the addition of $\frac{1}{2}QRI$ might result in a new Querier elected a bit too late and the flow of routed multicast traffic interrupted for a short while. Here is how that can happen. Suppose that initially there are one listener and two routers on a link, one router being the current Querier and the other, non-querier. When the Querier bails out, the non-querier will wait for up to $OQPT$ seconds before starting to send its own Queries. Next, the listener has right to delay its Report for up to QRI seconds. Then the Report can come late, that is, $OQPT + QRI - MALI = \frac{1}{2}QRI$ seconds after the multicast group ceased to be forwarded into the link due to a *MALI* timeout. A better choice for $OQPT$ can be just $QI \times RV$.

When a multicast router is just booting up or participating in an election, it needs to make sure that its Query reaches the other routers as well as the listeners. For that, the Query is repeated a few times (by default RV times [191, Section 9.7]) with pauses in between (by default $\frac{1}{4}QI$ [191, Section 9.6]).

Once the election is complete, all routers but the Querier go silent and start just watching MLD messages from other nodes. For this scheme to be stable, the state of each non-querier must be synchronized with that of the Querier with respect to the controlling records and timers. This synchronization has to start with the Querier advertising its current QI and RV values because the length of the $MALI$ interval depends on them. These values are denoted as **QQI** (Querier's QI) and **QRV** (Querier's RV), respectively, to emphasize they come from the Querier.

Consequently, the MLDv2 Query format must provide room for this information as shown in Fig. 6.10. The QRV value can be stored simply as an unsigned integer. The corresponding field is known under the same name, **QRV** [191, Section 5.1.8]—it was shown in Fig. 6.10. At the same time, the QQI value, if stored as a 16-bit unsigned integer number of milliseconds, can count only up to 65.535 seconds; and millisecond accuracy isn't really necessary over its entire range. A new code therefore will be required, to be "designed" as soon as we have finished with the present matter. The same code will come handy for storing a Maximum Response Delay value. The field where a QQI value is stored in this code is known as **QQIC** (Querier's Query Interval Code) [191, Section 5.1.9].

Next, when a multicast group is subject to a fast check with a Query, the following parameters come into play: the *LLQI* interval, the *LLQC* retransmit count, and the *LLQT* timeout. These parameters also need to be harmonized across all routers so that a non-querier won't remove a record or switch filter mode prematurely. Fortunately, our task is made easier by the fact that these parameter values won't really need to be advertised separately. Here is a possible rationale for that:

- The recommended value for *LLQC* is the same as *RV*, and the latter is already advertised as **QRV**.

- While performing a fast check, the Maximum Response Delay value is equal to *LLQI*, and there already is a field for the Maximum Response Delay in the Query format. So the non-queriers just need to put this equality back to front and use the Maximum Response Delay value received for their current *LLQI*.

- The *LLQT* value can be calculated from *LLQI* and *LLQC* as follows: $LLQT = LLQI \times LLQC$.

We aren't just speculating here; these considerations are used by existing implementations. For example, the MLDv2 module from *XORP* [200] sticks to them: It sets its *LLQC* from *QRV* and *LLQI* from the Maximum Response Delay if the Query received has a non-zero Multicast Address in it.

But how are the non-queriers supposed to detect that a fast check is in progress? This information can be inferred from the Query type as follows:

- If it is a General Query, i.e., one with a zero Multicast Address and an empty source list, it was not a fast check but an initial or periodic querying. Only the *RV* and *QI* values are to be taken from such a Query.

- If it is a Multicast Address Specific Query, having no sources listed in it, we are dealing with a fast check for remaining listeners of the group in *EXCLUDE* mode. In this case the Filter Timer of the group needs to be lowered to *LLQT*.

By Postel's Law, a non-querier must still check first that its own filter for this group is in *EXCLUDE* mode.

- If it is a Multicast Address and Source Specific Query, it was sent to perform a fast check of the specified sources sending to the given group. This is a cue to a non-querier that the timers on all those source records need to be lowered to *LLQT*.

> It can be a good idea for each non-querier to verify that the Query is coming from the current Querier and not from some rogue node. A meticulous implementation can try to keep track of the current Querier's IP address, updating it if a Query from a numerically smaller address is received. However, it will also have to handle an unexpected departure of the Querier. In practice it can be sufficient to compare the Source Address of the Query with the non-querier's own address and disregard the Query if the latter is less than the former. The rest of the stabilization job will be done by the election mechanism, which goes a long way to guaranteeing that there will be just one Querier ultimately left to advertise its parameter values.

This draft scheme for how non-queriers will pick up parameter values from the Querier looks really nice on paper, but it overlooks the possibility of packet loss. Alas, now the evil Reality Fairy is about to intrude and muddle up our elegant solution, but we can hardly help it.

As a matter of fact, the Querier can never know if the Query sent has been received by all non-queriers. There is no feedback from them—and there will never be one just to keep the complexity of the protocol in check. So the only means available to the Querier is to repeat its Query more than once in the hope that the chance of its reaching all non-queriers will be high enough. If periodic Queries don't need to be retransmitted as they are already being repeated at intervals of QI, fast-check Queries need to be retransmitted $LLQC - 1$ times each. This behavior isn't completely new to us because the same is necessary to make sure the Query reaches all listeners. However, the requirement to support non-queriers complicates this scheme even further.

Suppose that initially there is just one multicast router on a given link, so it is the Querier. In this case the router can stop its querying attempts as soon as the first Report about the group and the sources being checked comes through. Indeed, why to keep repeating the Query if all information needed has been received? Now let's introduce one or more non-querier routers on the link. For them all to receive all information, the Querier can no longer stop at the first Report received because some of the non-queriers might have missed some messages. Acting by the model that repeating is the way to reliability, the Querier must carry on sending copies of its Query until *LLQC* copies have been sent in total, and not a copy less.

What problems will this requirement cause to us? According to our scheme, each Query instance controls the timers of non-queriers, lowering them to the short interval $LLQT$. This makes the following scenario possible. Suppose there are a Querier, a non-querier, and a listener operating on a link; assume $LLQC = 2$.

1. The Querier sends the first copy of its Query $Q(G, S)$.

2. The non-querier lowers the timers on the source records from set S to $LLQT$.

3. After a short delay the listener responds with a Report confirming its interest in the sources from set S.

4. The Querier and the non-querier both receive the Report and reset the timers on the source records from S to the long interval $MALI$.

5. The Querier repeats its Query $Q(G, S)$.

6. The non-querier lowers the relevant timers again.

7. The listener fails to respond or the Report is lost.

8. The non-querier detects a timeout on the sources from S and blocks them.

9. The Querier and the non-querier are now out of sync!

A similar scenario can be constructed for the case of a Multicast Address Specific Query, $Q(G)$. In it, the Filter Timer will expire in the non-querier.

Unfortunately, the Querier has to work with incomplete information and so it can't really guarantee that the non-querier still receives all data even in the face of a brief network fault. Perhaps the best the Querier could do in step 5 of our scenario was indicate somehow to the non-querier that the timers need not be lowered again because the sources from set S are now known to be allowed on the link. This information can be encoded by a single flag bit in the Query format, whose meaning will be whether the non-queriers need to lower the relevant timers. It appears in what formerly was a reserved field, so its value of 1 will request the non-default behavior, that is, that the timers be not lowered. Hence its name: Suppress Router-Side Processing, or **S**-bit for short [191, Section 5.1.7].

How will the Querier detect it is time the **S**-bit were set to a value of one in the subsequent Query copies? If the listeners have confirmed their interest in all sources from set S, it will be sufficient just to set the **S**-bit to

1 in the in-memory image of the Query message kept by the Querier. Then all subsequent copies of the Query sent on the wire will have that bit set so. However, in reality only some subset $S_1 \subset S$ may get confirmed by the listeners. How can sets S_1 and $S_2 = S \setminus S_1$ be separated when the next copy of the Query is due to be sent? This is where the current timer intervals can come handy. The timers on the confirmed subset S_1 have been reset to the long interval $MALI$ whereas the timers on the yet-to-be-confirmed subset S_2 still have no interval longer than $LLQT$. Since there is just one **S**-bit per Query message, the original Query will have to be split into two messages, one with the source set S_1 and the **S**-bit of 1, and the other with the source set S_2 and the **S**-bit of 0.

The **S**-bit will also be meaningful in a Multicast Address Specific Query. As we know now, such a Query is sent to confirm *EXCLUDE* mode of a router filter. Since the router can't be halfway between *INCLUDE* and *EXCLUDE* modes, message splitting will never be required in this case. Still, the criteria when to flip the **S**-bit can remain essentially the same, i.e., based on the remaining interval of the relevant timer, which is the Filter Timer in this case: If the Filter Timer is still $LLQT$ or less, the **S**-bit must be zero; and if the Filter Timer has become greater that $LLQT$, it is time to toggle the **S**-bit to one.

So now the non-queriers will lower their timers not on any Query but only on a Query with $S = 0$. This prompts us to revise the timer lowering rules for the Querier as well. If there were no non-queriers at all, the Querier would be able to lower its timers just once, in the beginning of the Query sequence. In fact, our preliminary calculation of the $MALI$ and $LLQT$ timeouts was based on this simplified model. On the other hand, if non-queriers existed but packets were never lost, the Querier would pull the same trick of lowering the timers just once and so it would set the **S**-bit to 1 in all Query message copies but the first one in the sequence, for the non-queriers to do the same thing with respect to their own timers. That is, the first Query copy would cause the relevant timers of all routers to be reset to $LLQT$ while the subsequent copies would have no effect on the timers. In reality, however, any Query message can be lost. This is why the Querier must not flip the **S**-bit to one until it gets acknowledgement that the latest Query message has been received by at least some nodes, which have to be listeners due to no feedback from non-queriers. For the sake of keeping sync with the non-queriers, the Querier will have to lower its own timers to $LLQT$ each time a Query with $S = 0$ is sent out.

Granted, the trick just discussed will hardly help to make MLDv2 timer management more straightforward, but it can ensure a better synchronization between the multicast routers of the link. On the one hand, a source record can now live longer than necessary. For example, if there are no more listeners interested and so no Report is heard back, the record will live for *LLQT* seconds since the last Query in the series although by our original plan it would expire in the minimal time of *LLQI*. On the other hand, now there is a greater chance that the record will expire on all of the routers simultaneously.

The final rules for the Querier and non-queriers to keep sync with each other are as follows:

1. Building a Query—for the Querier only [191, Section 7.6.3]:

   (a) For a Multicast Address Specific Query: If the Filter Timer on the group record is *LLQT* or less, the **S**-bit must be set to 0. Otherwise it must be set to 1.

   (b) For a Multicast Address and Source Specific Query: If the timers on all of the source records are *LLQT* or less, the **S**-bit must be set to 0. If all of the timers are greater than *LLQT*, it must be set to 1. In a mixed case, the Query has to be split in two messages, one with $S = 0$ and the other with $S = 1$.

2. Sending or processing a Query—for all multicast routers [191, Section 7.6.1]:

   (a) If $S = 0$:

      i. For a Multicast Address Specific Query: The Filter Timer on the group record must be lowered to *LLQT*.

      ii. For a Multicast Address and Source Specific Query: The timers on the source records listed in the Query must be lowered to *LLQT*.

   (b) If $S = 1$, the timers are just left running on.

These rules indeed can help the synchronization between the Querier and non-queriers to cope with a brief network fault. For example, if the very first Report from the listener in the above scenario (page 344) reaches the non-querier but not the Querier, the non-querier will reset its timers to the long interval $MALI$ while the Querier's timers will be left counting down the short interval $LLQT$. However, the Querier will repeat its Query with $S = 0$ and the non-querier will lower its timers again, thus bringing the timers back into sync.

We are almost there now. Still, before the new version of MLD can be rolled out, we ought to take care of the most unpopular facet of protocol design: backward compatibility with the previous version.

Compatibility between network protocols has two basic aspects. One is the compatibility of the protocol message formats, while the other is a compatible behavior of the protocol parties as triggered by incoming messages. These aspects are closely related, of course.

The compatibility of message formats means, in essence, their unambiguous interpretation when they coexist on the wire. For instance, should we change the encoding of the **Maximum Response Delay** field for the MLDv2 Query format, we will need to ensure that an MLDv1 listener won't misinterpret it. In fact, this change still is on our to-do list and now is a good time to implement it as an exercise in providing protocol compatibility. We need a better code for the values of Maximum Response Delay and QI, and the new code must be backward-compatible with a fixed-length unsigned integer.

Here is how this compatibility can be provided: Let the most significant bit indicate the encoding used. If it is zero, the rest of the field is just an unsigned integer. If received by an old implementation, the most significant bit will be regarded as part of the integer field but it won't change its numeric value due to being zero. As long as an MLDv2-capable router is in compatibility mode, it must stick to this sub-code. Eventually, when compatibility with old implementations is no longer an issue, it will be safe to switch to an alternative code, to be indicated by the most significant bit set to one.

The new code needs to be able to represent a wide range of values with a reasonable accuracy. A well-known solution to this problem is a floating-point code, which borrows from scientific notation of numbers. In this code, a number is represented as an optimal mantissa times a power of the radix, which is 2 in this case: $x = M \times 2^E$. Here M is the fixed-length mantissa and E is the exponent.

A fixed-length mantissa will be optimal if all of its bits are significant. Such a mantissa is also known as normalized. To put it simply, there should be no leading zeros in the mantissa because they carry no information. Consequently, the high-order bit of M must be 1, otherwise it wouldn't be significant. Thanks to that, the normalized mantissa's high-order bit can be just implied rather than stored explicitly. Thus the overhead of the encoding flag is made up for.

> Although the traditional normalized form puts the radix point after the most significant digit of the mantissa, for our current purpose it will be handier to put it after the least significant digit so that the mantissa remains integral. Quite obviously, these two representations can be mapped to each other by a mere shift, that is, by changing the exponent E by the number of mantissa bits stored if the high-order bit is implied. E.g., the binary mantissa 1.0001 is converted to 10001 by subtracting 4 from the exponent; the value of the encoded number thus remains the same.

Let such a code be applied to the 16-bit field carrying the Maximum Response Delay value. In MLDv2 this field is called **Maximum Response Code** [191, Section 5.1.3], probably for its non-intuitive encoding; its place in the MLDv2 Query format was shown in Fig. 6.10. Let the exponent occupy 3 bits. One more bit has been used up by the encoding flag. So the mantissa can be 12 bits long. However, the values 0 through $32,767$ can already be represented in the old, unsigned integer code, so we are interested in representing the values from $32,768 = 2^{15}$ on. They will never have an exponent less than

$$\log_2 \frac{2^{15}}{2^{12}} = 3.$$

So the exponent can be offset by 3 for a greater range. The binary values of the mantissa, M', and the exponent, E', as stored in the **Maximum Response Code** field, can be just unsigned integers. This way the actual Maximum Response Delay value, if in the new code indicated by the high-order bit set to 1, will be as follows:

$$(0\text{x}1000 + M') \times 2^{E'+3}.$$

> Calculate the range of the new code for **Maximum Response Code**. (Hint: $32,768$ *through* $8,387,584$.)

Essentially the same code will be used to store a QQI value in the **QQIC** field, which stands for Querier's Query Interval Code [191, Section 5.1.9] and

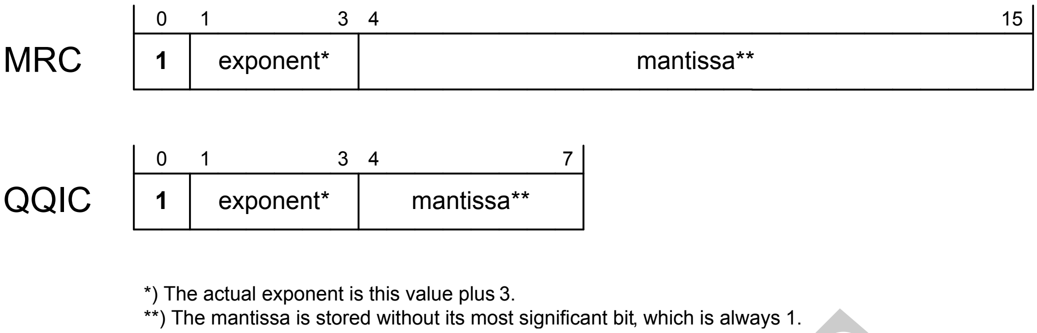

MRC
0
1
3
4
15
1
exponent*
mantissa**
QQIC
0
1
3
4
7
1
exponent*
mantissa**
*) The actual exponent is this value plus 3.
**) The mantissa is stored without its most significant bit, which is always 1.

Figure 6.11: Floating-point code for **Maximum Response Code** (MRC) and **QQIC**

was shown in Fig. 6.10. The only difference to allow for is that it is an 8-bit field, so the encoding flag and a 3-bit exponent leave room for a 4-bit mantissa. Consequently, if the field's high-order bit is set to 1, its numeric value will be as follows:

$$(0\text{x}10 + M') \times 2^{E'+3},$$

where M' is the mantissa value and E' is the exponent value as they are stored in the **QQIC** field.

Calculate the range of the new code for **QQIC**. (Hint: 128 *through* 31, 744.)

The extended, floating-point formats of the **Maximum Response Code** and **QQIC** fields are shown in Fig. 6.11.

The compatibility of individual fields ensured, we can move on to the issue of compatibility between the message formats as a whole. The MLDv2 Report format (see Fig. 6.7) is radically different from its MLDv1 counterpart (see Fig. 6.6) because it is a container for a variable number of multicast address records, each record being of a certain type and possibly having a list of sources in it. All we can do here in order to provide compatibility, as in unambiguity of the format, is keep the MLDv1 and MLDv2 Reports completely separate. For that, the MLDv2 Report just requires a new ICMPv6 Type assigned to it, which will be 143.

As for the MLDv2 Query format (Fig. 6.10), it has remained conforming with the general scheme of MLD where any extended fields are to be placed after the fixed, or at least familiar, part of the message. Thanks to this, the MLDv2 Query can have the same ICMPv6 Type as the MLDv1 Query, 130.

Table 6.7: Listener compatibility chart

Listener version	MLDv1 Query	MLDv2 Query
MLDv1	Not an issue	**What to do?**
MLDv2	Send an MLDv1 Report	Not an issue

Table 6.8: Multicast router compatibility chart

Router version	MLDv1 Report or Done	MLDv2 Report
MLDv1	Not an issue	Will ignore
MLDv2	**What to do?**	Not an issue

If an implementation supports both versions, it will be able to detect the version of an incoming Query message by its length: An MLDv1 Query is always 24 bytes long whereas an MLDv2 Query consists of 28 bytes or more. This leaves the lengths from 25 through 27 bytes invalid and corresponding to no MLD version.

> Note that the length of an MLDv2 message, be it a Query or a Report, is encoded in the message itself. For this reason, it will be possible to detect a future version of MLD only after parsing the MLDv2-compatible part.

At last we can discuss a compatible behavior of MLD parties. The low-hanging fruits here are the reaction of MLDv1 listeners to MLDv2 Queries and the reaction of MLDv2 routers to MLDv1 Report or Done messages. This can be noticed from the compatibility charts, Table 6.7 and Table 6.8.

> An MLDv2-capable listener must never send an MLDv2 Report in response to an MLDv1 Query at least because it will be just ignored due to an unknown ICMPv6 Type value.

In the first case the MLDv1 listener will respond to the MLDv2 Query with one or more MLDv1 Reports, depending on whether the Query was specific or general. Then the MLDv2 router that sent the Query will be able to detect that there is at least one MLDv1 listener on the link, and so the router will turn on compatibility mode.

This naturally leads us to the second case, that of an MLDv2 router getting MLDv1 Report or Done messages. A more advanced device, the MLDv2 router should be able to decode MLDv1 messages with ease. So the issue is

Table 6.9: MLD message upgrade chart

MLDv1 message	MLDv2 equivalent
Report	$\mathit{IS\_EX}(\emptyset)$
Done	$\mathit{TO\_IN}(\emptyset)$

how hard its compatibility mode will be to implement. In particular, will a separate code path be needed to react to such messages? Fortunately, there is an opportunity for code reuse as the MLDv1 Report and Done are effectively subsets of what information can be represented by the flexible MLDv2 Report. We have already discussed this matter, so that just a relevant "upgrade chart" is left to be provided now, as shown in Table 6.9.

Having "upgraded" an incoming MLDv1 Report or Done message internally, the MLDv2 router can process it by the rules formulated for MLDv2 Reports.

> Of course, the MLDv2 router will have to take into account that the Destination Address of an MLDv1 Report or Done message will be the multicast address reported instead of *FF02::16*. Still, this should cause no trouble to the router. As we discussed earlier in this section, a multicast router is supposed to listen to all multicast addresses in the background anyway, relying on a Router Alert option instead of the Destination Address to pick out interesting packets.

Besides the two obvious cases, we need to address the possible "cross-pollination" between listeners as well as between routers. As we know, in principle listeners can hear MLD Reports while routers can receive MLD Queries. Fortunately for us, MLDv2 Reports are sent to a dedicated multicast address normally reserved for routers, and so listeners of neither version are supposed to see MLDv2 Reports in the first place. As for a possible reaction by an MLDv2 listener to an MLDv1 Report, report suppression has been deprecated in MLDv2 and it is safe just to do nothing. These rules for listeners are summarized in Table 6.10.

> Still, it isn't forbidden for an MLDv2 listener to suppress its own Reports if receiving MLDv1 Reports from another listener [191, Section 8.2.2].

At the same time, a router must pay attention to other routers' Queries because that's the basis for the election of the Querier and the subsequent

Table 6.10: Listener “cross-pollination” chart

Listener version	MLDv1 Report or Done	MLDv2 Report
MLDv1	Not an issue	Shouldn’t receive
MLDv2	Can ignore	Shouldn’t receive

synchronization with it. This case is the trickiest one because the two MLD versions use quite different rules in this area. In particular, an MLDv1 router will be unable to recover the information necessary for its timer synchronization from an MLDv2 Report.

For these reasons and also for the sake of network stability, let’s regard a configuration where both MLDv1 and MLDv2 routers are present on one link as undesirable. To be precise, let an MLDv2 router fall back to sending MLDv1 Queries as long as there are MLDv1 routers on the link [191, Section 8.3.1]. This will be a reasonable compromise since typically the routers are fewer than the hosts while replacing them is easier. If upgrading the hosts is usually a complex, multi-stage process requiring a lot of careful planning, the subnet[7] can be upgraded literally in one go, e.g., by deploying a few up-to-date multicast routers on the link and temporarily shutting down the old ones. Later the old routers can be upgraded and put back into service one by one.

Propose a solution or workaround for the following problem. Suppose the last remaining MLDv1 router has just been removed from a link. So the MLDv2 routers already there are still in compatibility mode and sending MLDv1 Queries only. Each MLDv1 Query sent so makes the other routers believe there still is an MLDv1 router on the link. As a result, the MLDv2 routers will linger in compatibility mode indefinitely even though all MLDv1 routers are gone now. How can the MLDv2 routers get out of compatibility mode automatically? (Hint: *The RFC suggests manual intervention.*)

The compatibility between MLD parties of different versions can be summarized as shown in Table 6.11.

Our work on MLDv2 finished, let’s compare its features with those of MLDv1 in Table 6.12.

The last major question regarding MLD for us to consider will be its security [191, Section 10]. How well is this protocol protected from attacks

[7] In the old sense of this term: the forwarding plane of a network [49].

Table 6.11: MLD compatibility chart: Messages of which version to send

Hosts ↓ / Routers →	v1 only	v2 only	v1 and v2
v1 only	Use v1	v2 Queries, v1 Reports	Use v1
v2 only	Use v1	Use v2	Use v1
v1 and v2	Use v1	v2 Queries, v1 and v2 Reports	Use v1

Table 6.12: MLDv1 and MLDv2 feature table

	MLDv1	MLDv2
ICMPv6 message Types	Query (130); Report (131); Done (132)	MLD Query (130); MLDv2 Report (143)
Query variants	General Query; Multicast Address Specific Query	General Query; Multicast Address Specific Query; Multicast Address and Source Specific Query
Report structure	One group per message	Multiple groups per message; a list of sources for each group
Destination Address for Report	Group reported	All MLDv2-Capable Routers (*FF02::16*)
Querier election	By IPv6 address	By IPv6 address
Redundant report suppression	Yes	Considered harmful
SSM and SFM support	No	Yes
Timer interval coding	Integer	Floating-point, with backward compatibility

and how dangerous can such attacks be? Unlike ND, the state of an MLD party depends only on what MLD messages it has received and is unaffected by other external events.

> By contrast, the ND state can be altered by virtually any IPv6 packet. For example, a unicast router invokes its ND module on each transit packet received.

So the only attack vector against MLD is message forgery. The most basic means to protect a link from such attacks will be to insulate the link from any MLD messages coming from elsewhere. Indeed, MLD is an absolute link-local protocol by design. To ensure it stays confined to the link, all of the stock tools can be used. First of all, on output the Source Address of an MLD message must be link-local, except in a Report, where it may be unspecified. Still, an MLD Report with an unspecified Source Address can be good only for giving the smart LAN switches a hint before doing SLAAC or DAD, when its sender has no IPv6 unicast address yet. So the other IPv6 nodes can just ignore such a Report if received [191, Section 5.2.13]. Indeed, the ND messages, which constitute SLAAC and DAD, aren't routable anyway, so that the multicast routers don't need to care about listeners' membership in solicited-node multicast groups or the All Nodes of the Link group. Later, once the node has assigned a link-local address to its interface, the MLD Report can be repeated from that address. Therefore *on input the Source Address of any MLD message is required to be link-local*; otherwise the message must be ignored.

Next, all MLD messages are supposed to carry a Router Alert option, which facilitates picking such messages out. So any MLD-capable node (that is, any compliant IPv6 node) must double check that each MLD message received has this option in it.

Routers are additionally forbidden from forwarding MLD messages to anywhere. Forwarding them to another link is insecure whereas returning them to the same link can result in a multicast storm when multiple routers are involved and so the number of packets is allowed to grow exponentially in each round. For this reason, MLD messages get no GTSM protection; instead, they have their Hop Limit set to just 1. This is an extra precaution against forwarding an MLD message into the same link, which is generally allowed by the IPv6 scoped address architecture even in the link-local case (see Section 2.4).

Off-link attacks thwarted, let's turn to on-local threats, which are expectably more dangerous and harder to withstand. The main threat here comes from forged or just unauthorized MLD Queries. To start with, an attack can be mounted on the Querier election procedure. For that it is enough to start sending Queries from an IPv6 link-local address whose numerical value is as small as possible, e.g., *FE80::* or *FE80::1*. After the hijacking of the Querier role or along with it, the attacker can also suppress any forwarding of multicast traffic into the link. For that it is enough to periodically send a Multicast Address Specific Query about a random group unlikely to be interesting to any listener. As it can be seen, in this case the legitimate routers will indefinitely remain non-queriers while the listeners never get a chance to report their current status. The legitimate routers thus can be deceived into reckoning there are no listeners on the link.

At first glance, such an attack can be counteracted with a modified election procedure, in which the *OQPT* timer is reset only on the reception of a General Query. However, the attacker can respond to that by sending a General Query to the All MLDv2-Capable Routers address, *FF02::16*. The routers will have to accept and process such a Query as usual [191, Section 5.1.15]. Alright, we can imagine that we went as far as changing the protocol so a General Query *must* be sent to All Nodes, *FF02::1*. Then the attacker will play her ace and send the apparently valid Query packet encapsulated in an *Ethernet* frame with a MAC Destination Address of *33-33-00-00-00-16*. Due to multicast group pruning at the link layer, this frame will be likely to be forwarded by the smart switched LAN only to the MLD routers [201].

A practical countermeasure against this kind of attack is to reject packets coming from *FE80::* because it is a reserved anycast address (see Section 6.1) and to assign the numerically smallest addresses possible to the legitimate multicast routers: *FE80::1*, *FE80::2*, and so on. However, the attacker still can get a chance if the router *FE80::1* goes down, e.g., because of a DoS attack.

From a strategical perspective, an appropriate response to this class of attacks on MLD would be a system for certification of multicast routers similar to what we discussed regarding SEND in Section 6.3.

At the same time, a forged MLD Report is an oxymoron because joining a group or just starting to listen to it is free and available to any node on the link. A DoS attack still can be mounted by sending Reports about joining a huge number of groups, which can be covered up by spoofing the Source

Address. The first part of this sort of attack can be prevented by capping the number of groups that can be forwarded into the link—something to do anyway for stability reasons. As for source address spoofing, it isn't really critical because a listener's identity is irrelevant in MLD, albeit the spoofing can make tracking the attacker down a bit more difficult.

An unrelated yet important consideration is that allowing hosts to join arbitrary multicast groups can result in a leak of private traffic. To control group membership, a system for authenticating and authorizing MLD listeners would be required. An alternative to that is end-to-end security for IP multicast [202], which is in a better accord with the TCP/IP host philosophy: "Never rely on intermediate nodes if you can manage by yourself."

As insecure is multicasting that can be done by any source. SFM and SSM can't really help to secure it because the source address still can be spoofed. What can be of help here is the same end-to-end security system one might call "multicast IPsec".

6.5 To Fragment or Not to Fragment?

The IPv6 fragmentation mechanism introduced in Section 3.3.4 requires more commentary because it is significantly different from its IPv4 counterpart. The major difference is, of course, that IPv6 fragmentation can be performed only by the source node although the information about the optimal packet size is hidden further downstream in the network.

The first to be affected by the new fragmentation semantics are transport protocols. Let's start with the textbook case of TCP. Formerly, in IPv4 environment, TCP had two basic options:

- To clear the IPv4 **DF** bit and create segments of arbitrary size, relying on the network to fragment them when and where necessary.

- To set the IPv4 **DF** bit and perform PMTUD in order to avoid IP-level fragmentation and optimize the TCP segment size.

A more sophisticated implementation could try the latter option but fall back to the former one if PMTUD proved broken, e.g., due to an overly strict filtering of ICMP in the network [203, Section 2.1].

In IPv6 the first option is no longer available because the **DF** bit is implicitly set in all packets and a packet exceeding its PMTU will never reach the destination. Now the packet's source must control its size appropriately. Consequently, *reliable operation of PMTUD becomes extremely important when transitioning to IPv6.*

This fact must be kept in mind by network operators eager to fight "undesirable" ICMP traffic.

Still, PMTUD won't be hindered by a faulty ICMP signaling if helped by the end-to-end principle[a] and some heuristic. For example, if a TCP three-way handshake was successful but full-size TCP segments are apparently lost as the far-end peer never acknowledges new data, this is very likely to be a PMTU related problem and the TCP sender can try reducing the segment size. Although there will be no ICMP hints about the next MTU size, a binary search can be nearly as efficient in providing a good approximation for the PMTU value. The binary search will be bounded by a known good value, such as 1280 bytes in IPv6, and the sender's interface MTU. The trial value can be increased as long as the segments are making it through. When the trial value exceeds the PMTU, the segments will be disappearing again and it will be time to start reducing the trial value instead. This elegant idea is at the heart of the so-called Packetization Layer PMTUD [204].

[a] "Rely on yourself before the far-end peer and on the far-end peer before the intermediate nodes."

Nevertheless, PMTUD isn't absolutely necessary even in IPv6 environment. Now that the minimum MTU size has become relatively large, 1280 bytes, a very basic TCP implementation can afford to use this constant as the one-size-fits-all PMTU estimate. This can also be the contingency plan for a more sophisticated implementation in case PMTUD is found nonfunctional after all.

TCP thus has two options when in IPv6 environment:

- To packetize the byte stream for a constant PMTU value of 1280 bytes.
- To optimize the segment size using PMTUD.

TCP was an example of a protocol able to fine-tune its segment size and so requiring no IP-level fragmentation. Block-oriented transport protocols and their consumers are an entirely different story.

Suppose some application protocol runs over UDP. In this case the datagram size is controlled by the application and can end up quite substantial. We aren't just making it up; there are real-world protocols such as NFS routinely creating datagrams a few thousand bytes long. How can such a datagram reach an IPv6 destination?

We have already discovered one possible solution to this. The application can request through the API that its datagrams be fragmented by the local IPv6 stack assuming a PMTU of 1280 bytes. This trick is simple and effective, if slightly suboptimal; it can ensure the packets sent on the wire won't be discarded downstream because of their size.

Still, packets can be lost in a network due to many reasons besides MTU issues, and an application protocol should be designed to cope with, or safely ignore, packet loss. This quality of UDP-based protocols can be employed to optimize the IPv6 fragment size. For this, the IPv6 layer itself needs to be doing PMTUD. This scheme can operate, e.g., as follows:

1. The applications sends its first large message to the destination.

2. The local IPv6 module fragments the resulting packet based on the egress interface MTU.

3. Quite probably, such fragments won't make it through the path, but in that case the IPv6 module will receive ICMPv6 Packet Too Big messages and update its PMTU estimate from them.

> This new estimate will be real because an ICMPv6 Packet Too Big message must include the MTU value that blocked the original packet. In the classic IPv4 a source would have to guess this value because it wasn't signaled back. Still, a binary search would converge fast enough for this to create no problem. Can you think of a scenario where a binary search for PMTU will outperform a search based on the MTU hints from the routers? (Hint: *One such case is a sufficiently long path where each subsequent MTU is smaller than the preceding one—can be visualized as a tapering path.*)

4. The application will either detect a lost message and retransmit it, or just ignore the loss and move on to sending the next message, depending on the protocol requirements.

5. The IPv6 module will fragment this packet using the updated PMTU value.

6. If the fragments still hit another small MTU and get dropped, the local IPv6 module will update its PMTU value again.

7. The application will retransmit the message yet again or send yet another message. . .

And so on, until the PMTU estimate falls down to the real PMTU of the network path the packets are taking. Once there, the fragmented packets will start reaching the destination and the session will proceed. For such a scheme to operate, the IPv6 module will need to cache the PMTU values learned per destination. The natural place to keep this sort of data will be the Destination Cache [205, Section 5.2].

> As we certainly remember, the path is the way through the network, from the source to the destination via the intermediate nodes, that a packet is taking, or would take, at a certain moment. For simplicity, we will disregard both source routing, where the path is selected by the source, and policy-based routing, where the next-hop decision is based on arbitrary metrics of the packet. Then a packet's path depends on the source node, the Destination Address, and the configuration of the source node and each subsequent intermediate node—roughly speaking, on their routing tables. Granted, in a live network the path a packet would take can change over time along with the routing tables, but when the network is stable the path remains the same at least for a while. This is what gives the source the right to cache PMTU values per destination.
>
> The mapping from a Destination Address to the observed PMTU value can become ambiguous if ECMP is in use and packets for the same destination can take different paths with inconsistent PMTU values. Keep in mind that the real-world IP host implementations have cached just one PMTU value per destination since the IPv4 era and IPv6 has only standardized this practice. Extra care therefore needs to be taken when using an ECMP hash function that depends on anything but the Destination Address, e.g., the upper-layer protocol or the TCP/UDP port numbers. (As an exercise, discuss how safe it is for an ECMP hash function to depend on the Source Address.)

Yet another option becomes available if the application protocol can vary its message size, for example, when the application is segmenting a stream of bytes or larger data units. In this case the application can try to pick such a message size that its IP fragmentation won't be necessary over the given

Table 6.13: Fragmentation and PMTUD related socket options

Option	Meaning
IPV6_USE_MIN_MTU	Just use the minimal MTU (1280 bytes)
IPV6_PATHMTU	Get the current PMTU for a connected socket
IPV6_RECVPATHMTU	Allow returning PMTU from *recvfrom()*
IPV6_DONTFRAG	Prohibit fragmentation of packets this socket sends

path.[8] Still, for the application to learn a suitable message size, a feedback channel is required from the IPv6 module to the application. It can be implemented within the network API as follows. The application queries the IPv6 module for the current PMTU value for the given destination. Alternatively, the IPv6 module can return the PMTU value back to the application in a control data structure[9] when the application is waiting for a response packet from the remote peer but an ICMPv6 Packet Too Big message comes through instead.

Lastly, an optimistic application that fully trusts PMTUD will want to prohibit any fragmentation of its outgoing packets via the API. Such an application will have to be constantly monitoring the current PMTU estimate and adjusting the size of its messages accordingly. This is how the IPv6 module and the application team up to perform PMTUD: The former processes ICMPv6 messages and updates the PMTU estimate while the latter generates new probe packets according to the latest estimate value.

These ideas have been reflected in the de facto standard Advanced Sockets API for IPv6 [206]. An application can control the fragmentation and PMTUD related aspects of a socket using the socket options from Table 6.13 at the *IPPROTO_IPV6* level [206, Section 11]. Some of those options can be specified for an individual message using an accompanying *cmsghdr* structure.

Discuss the pros and cons of an alternative approach to PMTU discovery where each intermediate node, when forwarding the packet, is to write the actual MTU value to a special IPv6 Hop-by-Hop option if its current value is greater than the MTU [207]. Thus a short probe packet would contain the exact PMTU value by its arrival at the destination.

[8] This technique is sometimes referred to as “semantic fragmentation” because it requires a certain insight into what the message data mean. We already met it in the context of MLDv2—see Section 6.4.

[9] In the *Berkeley Sockets* API such a structure is *cmsghdr*.

Another, more practical way to reduce the percentage of fragmented packets in a network is to avoid reduced interface MTU values as far as possible [82, Section 5]. Today's de facto standard for an end-user link is *Ethernet*, imposing a uniform MTU value of 1500 bytes. The same value has been adopted by other link protocols such as PPP [208, Section 2]. In a network core, extra headroom may be required to allow for complex encapsulation schemes such as tunnels without incurring fragmentation. Granted, it can be a non-trivial task to pick a good MTU value for each link when on an internetwork whose parts are managed by different entities, but it should certainly not drop below 1500 bytes in any event.

While at the topic of IPv6 fragmentation, there is one more standard value we need to specify: the minimal reassembly buffer size. As we may remember, in IPv4 it used to be 576 bytes. Now we can see a value that is going to suit the modern reality better: 1500 bytes. In other words, an IPv6 destination must be prepared to reassemble any packet with a final size of 1500 bytes or less [82, Section 5]. In turn, *an upper-layer protocol sender shouldn't create IPv6 packets longer than 1500 bytes unless the receiver has explicitly agreed to accept larger packets*. For example, TCP can convey this kind of information in an MSS option.

Calculate the most popular lengths of full-size TCP segments to be seen in an IPv6 network. Keep in mind that the actual length of a full-size TCP segment may not be as large as the receiver's MSS option value [209]. (Hint: *Allow for such TCP extensions as [210] and SACK.*)

6.6 Handling Multiple Local Addresses and Default Address Selection

As was said in Section 2.5, an IPv6 host can have multiple unicast addresses. In fact, it will have just one address only so long as it is completely disconnected: the loopback address *::1*. As soon as the host's interface other than the loopback gets activated, the number of addresses owned by the host will rise to two because that interface will need a link-local address to start with. To access the Internet, the host's interface will need a global address on top of that, bringing the total number of the host's addresses up to three. Thus three IPv6 unicast addresses are the bare minimum for an IPv6 Internet host.

However, even more than three addresses may be required by a host in certain scenarios. Suppose the host still has just one active interface. By the IPv6 scoped address architecture from Section 2.4, the interface belongs to zones of different scope, receiving a local address from at least some of them. Although in the modern practice we can encounter only the link-local scope and the global scope, the address architecture also provides for scopes of intermediate size and corresponding zones.

> As we learned in Section 2.4, an abstract, numerical IPv6 address comes from just one scope while an IPv6 address assigned to an interface belongs to a certain zone. At the same time, an IPv6-compliant interface belongs to exactly one zone of each possible scope regardless of what addresses are actually assigned to it.

Furthermore, a network interface can get more than one address from the same zone. What can this be good for? A typical case where multiple addresses from the same subnet are assigned to a host is when host B is taking over the services that were formerly offered by a now defunct host A. To minimize the configuration changes required in client hosts, A's address can just be reassigned to B; then B will start responding on behalf of A, and the client hosts will remain as happy with their existing configuration. As for addresses coming from different subnets, they can enable the host, for example, to make use of multiple upstream connectivity providers, i.e., to go multihomed.

> Of course, there is much more to multihoming than just giving the host several addresses. A review of this topic can be found in [211], while we are going to discuss it briefly in Section 6.8.

It isn't hard to anticipate that the situation will become even more complex if the host has more than one active interface because each of them will get, besides a mandatory link-local address, zero or more addresses according to the address plan of the network.

Now let's consider from this perspective a typical scenario where our local host L initiates sending an IPv6 packet to some remote host R. To put this in a more practical context, suppose an application running on host L wants to send a UDP datagram or establish a TCP connection to R. To be able to build such a packet, L's stack, not without help from the application or even the user, must render the abstract names "L" and "R" into concrete Source Address and Destination Address.

We are interested specifically in a case where host L has freedom to choose what addresses to put in the packet header. By contrast, were host L to respond to a packet from R, it would have little to no such freedom. In a traditional protocol the only "option" available to L would be to swap the Source Address and the Destination Address as selected by R; otherwise the response packet would be highly likely to be rejected by R. New-style protocols such as Shim6 or SCTP support more than one address on either session endpoint, but their choice of addresses for each packet to send is governed by the current reachability situation. At the same time, what we are interested in now is the initial choice based on a priori information such as the type or scope of an address.

As the very first step, the user is expected to supply the application with some machine-readable ID of host R. Today this niche is largely occupied by text-based hostnames, so the application will see such a name, e.g., *r.example.org*. By querying the DNS, this name will be resolved to a list of addresses. As was said in Section 2.11, we hope to see only global IPv6 addresses in public DNS zones; but there are also alternatives to the DNS, such as NIS, WINS, or even a *HOSTS.TXT* file... What are we getting at here? Speaking in general, the name resolution process can return a whole *set* of addresses reportedly belonging to host R, some of them probably having non-global scope.

For its part host L can have more than one local address to consider. If host L has N addresses while host R has M addresses, there will be $N \times M$ ordered pairs of addresses, each pair being a viable candidate.

In some cases the burden of the choice can be taken by the user or some external mechanism. Then the application will just use the API to request a specific address pair be used to speak to host R. For example, the *ping6* command supports that in its command line syntax:

```
ping6 -S FE80::1 2001:DB8::1
```

However, more often it is assumed that the application and the network stack are able to pick an optimal address pair, that is, to perform **Default Address Selection** [212] for a communication between the local host and the remote host.

In IPv4 such a selection procedure was quite rudimentary and boiled down to the application trying the remote addresses one by one. For instance, this is how a *telnet* client would do it:

```
$ telnet host.example.org
```

```
Trying 192.0.2.23...
telnet: connect to address 192.0.2.23: Operation timed out
Trying 198.51.100.7...
telnet: connect to address 198.51.100.7: No route to host
Trying 203.0.113.5...
Connected to host.example.org.
Escape character is '^]'.

NetBSD/vax (host.example.org) (ttyp0)

login:
```

The IPv4 stack on its part would pick a local address for the remote address passed down by the application. For example, the *BSD Unix* stack would use the routing table to look up the egress interface and then take its first address listed in the internal data structure [137, Section 22.8: in_pcbconnect Function]. This procedure would run until the communication was successful or the list of remote addresses was exhausted.

> The IPv4 default address selection procedure was in fact regulated by the standards [51, Section 3.3.4.3][213, Section 2.3], and the BSD stack behavior quite complied with them. Namely, the application would try the remote addresses in succession while the stack would pick the local address based on the egress interface of the packet or connection. For a TCP session this would be done just once when opening it so that subsequent routing changes didn't affect the socket's local address, which would have caused the session to fail.

Once each address is equipped with scope, such a naive address selection procedure as was employed by IPv4 can fail easily. Suppose that at the head of the interface address list is the mandatory link-local address, merely because it was assigned before the rest. Then host L would be able to speak only to its neighbors even if its interface also has a global address. This would be so because of the zone isolation principle from Section 2.4: *For an IPv6 packet to reach its destination, the destination interface must be in the source address zone.* To meet this requirement, IPv6 Default Address Selection will have to consider all suitable local addresses for each remote address.

> If no address choice can satisfy the zone isolation requirement, communication between hosts L and R will be just impossible. For example, this will be the case if host L has only link-local addresses while all of R's addresses are off-link (see Section 5.2). However, Default Address Selection can't know whether a remote address is on-link or off-link because this information is supplied by ND, which is invoked as late as when the packet is ready to go, whereas Default Address Selection runs when the packet is only being filled in. For this reason Default Address Selection can rely only on a priori criteria to choose an optimal address pair.

> The other half of the zone isolation principle states that an IPv6 packet can be delivered only if *the source interface is in the destination address zone.* Since an IPv6 address can't be qualified with a zone_id when published in the DNS, only its scope will be known, as was discussed in Section 2.11. So a host has right to assume that a remote address learned from the DNS is in the same zone of that scope as the egress interface. This is further supported by the fundamental postulate from Section 2.4 that *an IPv6 interface is in one zone of each possible scope.* For example, if the site-local address *FEC0::1* has been returned by the DNS, the host will conclude that the address comes from the local site rather than from elsewhere. This calls for extreme care when publishing non-global addresses in the DNS.

Now, what criteria are to be used when matching the local addresses with a remote address? In other words, which local address will be the best fit for a given remote address? To ensure such criteria can be cast in an algorithm, let's formulate them as a transitive "better/worse" relation between two source address candidates, SA and SB, when a destination address D is given. After that, it will be enough to sort the local address list using this relation for the best candidate to float up to the top.

> The sorting can be substituted by a mere linear search for the best element. A binary search won't quite work here because the order of the elements depends on a variable parameter, D, so the list can't be sorted once and for all.

First we will consider the case where the destination address scope is no larger than the scopes of the candidate source addresses: $Scope(D) \le Scope(SA) \le Scope(SB)$ or $Scope(D) \le Scope(SB) \le Scope(SA)$. When it is so, the chances of a successful communication are pretty good with either choice. Then it makes sense to prefer the candidate source address of the smaller scope because it will be closer to the destination address scope. For example, communication with a link-local destination address will be carried out using a link-local source address. This is good, for one thing, because the remote host's security policy can be more liberal to link-local addresses than to, say, global addresses.

If the destination address scope is intermediate between the scopes of the candidate source addresses, $\mathit{Scope}(SA) < \mathit{Scope}(D) \leq \mathit{Scope}(SB)$ or $\mathit{Scope}(SB) < \mathit{Scope}(D) \leq \mathit{Scope}(SA)$, the better chance is provided by the source address of the larger scope. At the same time, its competitor's chance isn't quite satisfactory because its scope is strictly smaller than that of the destination address, as their possible equality was handled under the previous case.

Lastly the destination address scope can be strictly larger than either of the candidate source address scopes: $\mathit{Scope}(SA) \leq \mathit{Scope}(SB) < \mathit{Scope}(D)$ or $\mathit{Scope}(SB) \leq \mathit{Scope}(SA) < \mathit{Scope}(D)$. Even in this case all isn't lost because the destination interface still can happen to be in the source address scope, e.g., when a global destination address is on-link. The chances of success will be maximized if the candidate source address of the larger scope is preferred because its scope potentially encompasses more interfaces. The host can do no better in this case.

To get a better handle on these rules, let's visualize them as an area graph using one axis for SA and the other for SB as shown in Fig. 6.12. With such axes, the 2D space of this problem's solutions can be divided into four areas according to the rules just worked out. In two of them the answer is definite while in the other two it depends on how the candidate source address scopes compare with each other.

If a line with a slope of 1 is additionally drawn through the graph's origin, the solution plane will be divided in two halves so that the top half has $\mathit{Scope}(SA) < \mathit{Scope}(SB)$ and the bottom half has $\mathit{Scope}(SA) > \mathit{Scope}(SB)$. This line will effectively help us to select the better source address where it is still conditional. So the graph can be reviewed to specify a definite answer in all areas as shown in Fig. 6.13.

At first glance, we have six areas now. However, their number can be reduced to just four because two SA areas share a boundary and so do two SB areas; in Fig. 6.13 those areas are circled. So the final revision of our graph will look as shown in Fig. 6.14.

Guided by the final graph, it is easy to work out the optimal algorithm [212, Section 5]:

```
If Scope(SA) < Scope(SB) then:
```

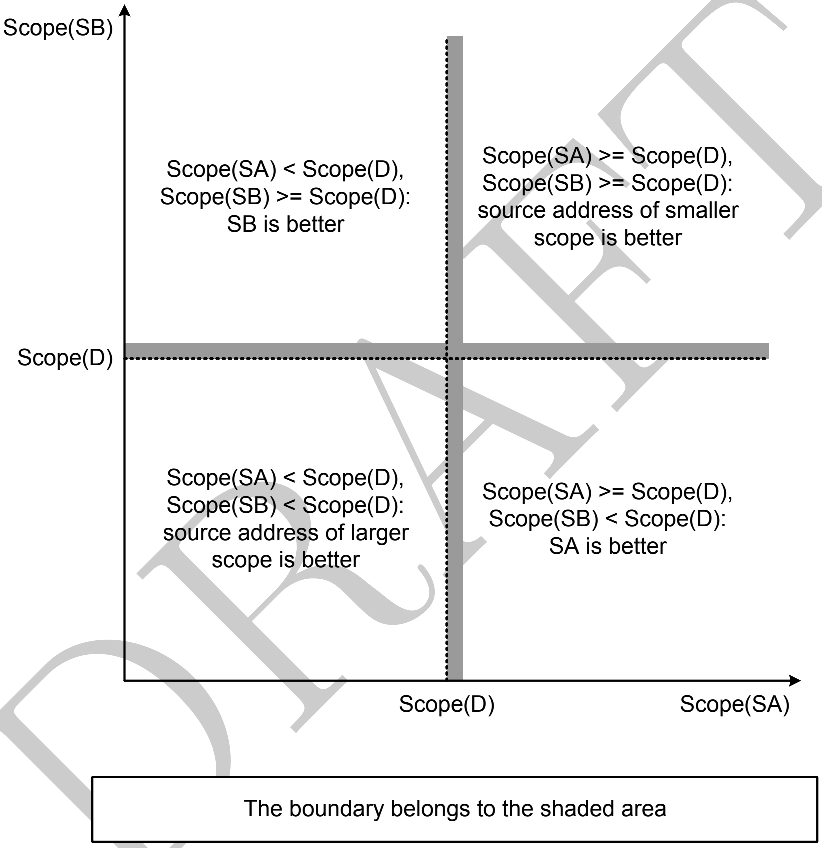


Figure 6.12: Selecting source address by scope: initial scheme

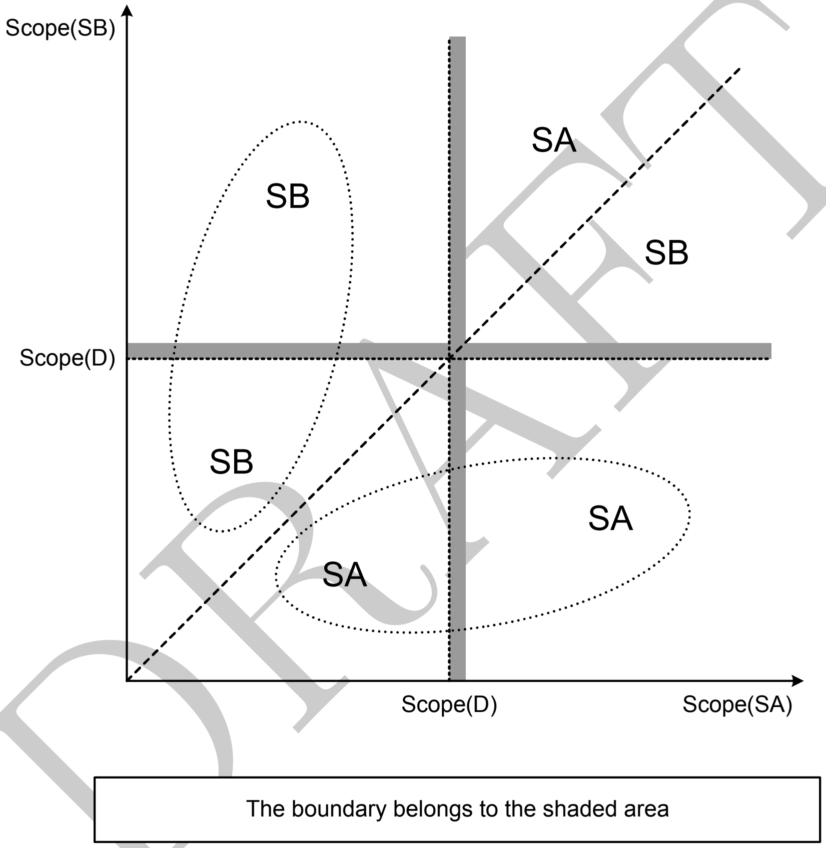


Figure 6.13: Selecting source address by scope: intermediate scheme

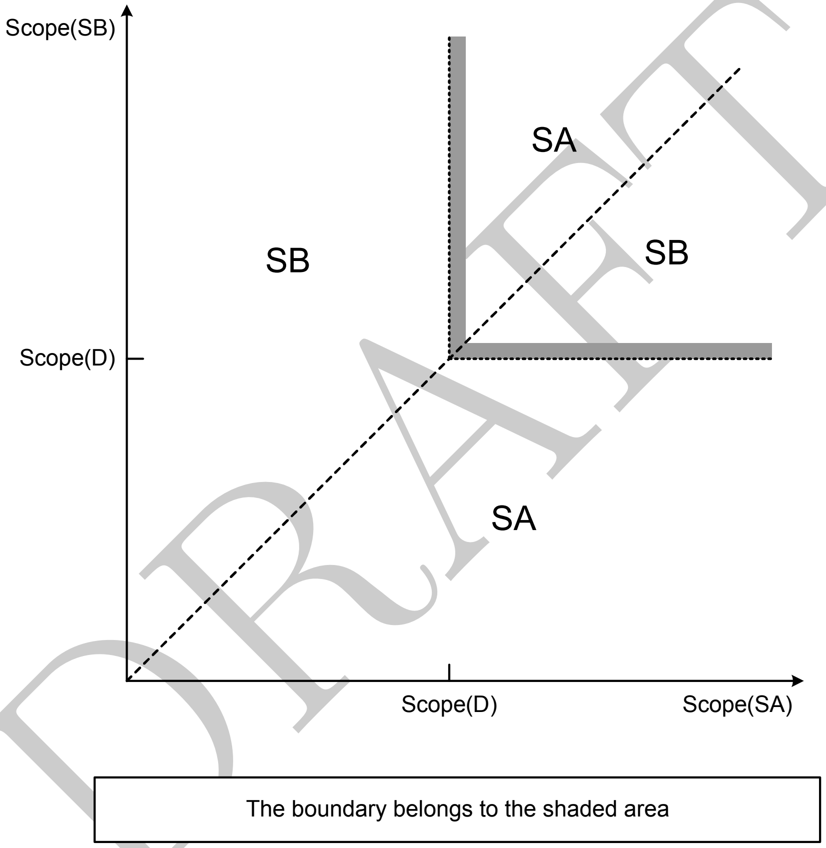


Figure 6.14: Selecting source address by scope: final scheme

```
    If Scope(SA) < Scope(D) then:
        Prefer SB.
    Else:
        Prefer SA.
Else If Scope(SA) > Scope(SB) then:
    If Scope(SB) < Scope(D) then:
        Prefer SA.
    Else:
        Prefer SB.
Else:
    Scopes of SA and SB are equal; answer undefined.
```

Verify that such a relation is indeed transitive, as in, if SA is better than SB and SB is better than SC then SA is better than SC.

What if the scopes of SA and SB are equal? In this case some additional selection criteria will have to be employed. One of them can be concerned with the lifecycle of a local IPv6 address, which was discussed in Section 5.4.2. Namely, once an address has become Deprecated, it may be selected only if there is no Preferred address of satisfactory scope available.

Another rule can be discovered if we try to answer the following question. Back in Section 2.5, we concluded that having only link-local addresses can be sufficient for a router to perform its primary function, that is, to forward transit packets. Indeed, as soon as each of the router's interfaces gets a link-local address, its neighbors on that link will be able to use it as a next hop. Besides, this will enable the router to participate in Neighbor Discovery (Section 5.1) and Router Discovery (Section 5.4.2), and also to be the MLD Querier (Section 6.4) because all these functions of a router simply must be performed from a link-local address.

There is at least one case though where the router may need an address of larger scope. Suppose the router attempted forwarding a packet only to find that it exceeded the egress interface MTU. Now the router must discard the original packet and issue a Packet Too Big error notification to the original packet's Source Address, which can be of any scope, up to global one.

It isn't hard to see what we are driving at. Even a router sometimes has to do a host's job when originating an ICMPv6 packet. Its Destination Address is taken from the Source Address of the offending packet, but its Source Address choice is up to the router. For the ICMPv6 packet to be able to reach its destination, its Source Address must be of appropriate scope. For example, if the offending packet was sent from a global address, the ICMPv6 packet will generally need a global Source Address as well. Otherwise the ICMPv6 packet will reach its destination only in some special cases as was discussed in Section 2.4, while we are speaking of the most general case now.

How can the router have a global address for this purpose? Will a global subnet have to be assigned to each of its interfaces? That won't be required if the router has just one global address, e.g., on its loopback interface, providing it can be included in the list of candidate source addresses when sending a packet via any interface. Then the main selection rule concerned with scopes will automatically select the global source address.

A normal host equipped with multiple interfaces might also benefit from the right to consider all of its addresses, although it is recommended for such a host to limit the candidate list to the egress interface addresses only [212, Section 4].

> We say "host" rather than "node" through this section to emphasize that creating new packets is essentially a host's job. The other of its jobs is to consume packets addressed to it. At the same time, these are untypical functions for an ideal router as it should be concerned only with forwarding somebody else's packets. Only due to how IP control signaling works even a most basic router has sometimes to do a host's job.

On the other hand, if the egress interface happens to have an address of suitable scope, it should be preferred. This is required, for one thing, for correct operation of Redirect. The issue is that a router can issue a Redirect only if the Source Address of the original packet is on-link. So let this be a lower-precedence rule: If one candidate source address belongs to the egress interface and the other comes from a different interface, the former wins.

One more rule is needed to take into account that host addresses are classified as public or temporary by Privacy Extensions from Section 6.2. Privacy Extensions must be disabled by default, so having them enabled on a host is a clear sign the host's operator actually means to use them. For this reason a temporary address will be preferred over a public address. Still, the API must provide means to request a public address instead.

Lastly, if the routing hierarchy and the address assignment hierarchy coincide, the length of the common prefix in two addresses can indicate the distance between the interfaces those addresses are on: the longer the common prefix, the fewer hops in the path, provided the addresses come from the same zone. So there is sense in preferring the candidate source address having the longer common prefix with the destination address D, at least if the stronger criteria produced a tie. For example, if only SB comes from the same subnet as D, SB should be preferred over SA.

The common prefix lookup should be limited to the subnet prefix of the local address because comparing the interface ID bits won't produce the information required. Roughly speaking, if the first 64 bits of the addresses being compared are the same, there is no sense in comparing the rest of their bits.

It is these criteria put in this order of precedence that form the core of the source address selection procedure [212, Section 5]. Let's recap them:

1. Most suitable scope.
2. Preferred vs. Deprecated.
3. An address on the egress interface vs. the rest.
4. Temporary vs. public.
5. Longest common prefix.

This procedure will return the best local address for a given remote address D. Let's denote it by $Source(D)$.

The complete list of rules from [212, Section 5] is somewhat longer. At its head is the rule to prefer the source address equal to the destination address so loopback communication is supported. Indeed, the Source Address and the Destination Address being the same is the most reliable indication that the packet is being looped back to the local host. Other criteria are there to support IP Mobility and to allow affecting Default Address Selection decisions through labels and precedence values from a policy table. They are of undoubted practical importance, but we don't think their fundamental value is high enough to discuss them here. We are sure the reader will be perfectly able to make sense of them when necessary.

Now we should recall that the host has also learned a whole set of potential destination addresses. It is impossible to select just one of them and call it the winner because it is unknown in advance which of them are actually reachable under the present conditions in the network. So the host has to try them one by one until the communication is successful. Nevertheless, sorting the list of candidate destination addresses too is as good an idea, so that the addresses more likely to work float up to the top and are tried before the rest.

This means we also need to work out transitive criteria to compare two candidate destination addresses, DA and DB. To do that, we can build on the rules for source address selection because any remote addresses are seen by the local host through its own interfaces and in the light of its own addresses. So it is unsurprising that the destination address selection procedure [212, Section 6] will invoke the source address selection one, $Source(D)$, all the time and compare the metrics of the source addresses so returned to see which destination address of the two is better.

Let's start by supporting the scoped address architecture with the following rule: Of the two destination addresses compared, wins the one of the same scope as its corresponding source address. For instance, if $Scope(DB) = Scope(Source(DB))$ but $Scope(DA) \neq Scope(Source(DA))$, DB will win. Of course, this won't always give a definite answer because such scopes can be the same, or different, for both destination addresses considered.

Should the previous rule return a tie, the candidate destination address whose corresponding source address is Deprecated is to lose. For example, if $Source(DB)$ is Deprecated but $Source(DA)$ isn't, DA will win. This rule can also return a tie if both source addresses, $Source(DA)$ and $Source(DB)$, are in the same stage of the address lifecycle.

Next comes into play another rule concerned with scopes. The destination address scope can provide some hint as to how long and thorny the path to the destination interface is. For instance: if the destination address is link-local, the interface bearing it is supposed to be just one hop away; and if it is site-local, the path at least lies within one administrative domain and so it can be expected not to suffer from inter-domain configuration issues such as inconsistent border policies. So, other metrics being equal, it makes sense to prefer the candidate destination address of the smaller scope.

If there still is a tie, the last rule will be to prefer the candidate destination address having the longer common prefix with its corresponding source address.

And again we have skipped some of the selection rules [212, Section 6], focusing only on the most fundamental of them.

May there be any hidden inconsistencies in this procedure for Default Address Selection? One of its details likely to look fishy to a competent IPv4 engineer is the use of the packet's egress interface to work out the source address, which is done before the destination address is certainly known. The apparent problem is that, by the IPv4 rules, it is the egress interface that is to be worked out based on the destination address of the packet.

Of course, this problem could be worked around if the egress interface as considered by the $Source(D)$ procedure were made a function of the destination address D instead of a fixed value selected beforehand. If in the IPv4 reality, such a function would boil down to a routing table lookup.

Still, the IPv6 architecture is going to employ a quite different approach, which started to materialize back when we were busy working at ND (Section 5.1) and SLAAC (Section 5.4.2). Back then it was postulated that all structures of the reference IPv6 host would be bound to a specific interface while interface selection was to be performed outside of ND. This model is to be finalized in Section 6.8, and now we will just take another essential step toward it.

As was discussed at length in Section 5.2, the reference IPv6 host isn't going to be the same, with respect to its internals, as the traditional IPv4 hosts or routers. At the same time, we have no model for the IPv6 router yet. Let's assume that, internally, it won't be too different from its IPv4 cousin, its operation still to revolve around a routing table mapping variable-length prefixes to next hops.

As we have already said in this section, the only basic case where an IPv6 router will need to invoke Default Address Selection is when issuing an ICMPv6 notification message. In this case the router will hardly have a problem because the destination address need not be selected: it must be the same as the source address of the offending packet. So the router will just look the egress interface up in its routing table by the destination address and then perform source address selection.

> The zone of the ICMPv6 message destination is indexed by the ingress interface of the offending packet. Should the physical router comprise multiple virtual routers, that interface will also point to the virtual router whose routing table is to be consulted when sending the ICMPv6 message.

Multiple destination address choices therefore can be encountered only by a host. However strange may it sound to an IPv4 expert, the IPv6 host architecture we are to arrive at will allow the egress interface to be selected prior to routing, that is, before the next hop decision [212, Section 7]. E.g., an application may want to select just one of the host interfaces and have all its outgoing packets sent out through it.

Default Address Selection can cope with multicast destination addresses as well. Of course, in this case the candidate source address set has to be limited to the addresses from the egress interface, or from another interface into the same link, to meet the multicast routing requirements [212, Section 4].

A means must be provided for an application to control various aspects of Default Address Selection, e.g., to prefer public addresses over temporary ones. There is a standard API for this purpose [214].

What if the candidate destination address set as returned, e.g., by a DNS lookup contains both IPv6 and IPv4 addresses? It is nice to known that the standard Default Address Selection procedure can handle the mixed case gracefully, although we won't discuss it herein. Granted, the transition from IPv4 to IPv6 is a very important topic, but we believe that it shouldn't be studied before one has a clear idea of how IPv6 alone works.

Concerns have been raised that the current Default Address Selection procedure can't handle all imaginable cases well [215][216]. So it is still a research area.

For us to have a complete picture, let's now list all addresses an IPv6 host is required to recognize as identifying itself [21, Section 2.8]:

- The loopback address (see Section 2.3).
- Its required link-local address for each interface (see Section 2.5).
- Any additional unicast (Section 2.6) and anycast (Section 5.3) addresses that have been configured on the host's interfaces, manually or automatically (Section 5.4).
- The well-known All Nodes multicast addresses defined in Table 2.6.
- The solicited-node multicast address for each of its unicast and anycast addresses (see Section 5.1).
- Multicast addresses of all other groups to which the host currently belongs (Section 2.8).

An IPv6 router is required to recognize all addresses that a host would, plus the following addresses as identifying itself:

- The Subnet Router anycast addresses (Section 6.1) for all interfaces on which it is configured to act as a router.

- The well-known All Routers multicast addresses defined in Table 2.6.

6.7 Name Resolution in IPv6 Environment

Suppose the DNS servers and clients have adopted the basic changes discussed in Section 2.11. Then an application invoking DNS resolver functions from the standard library can expect to see IPv6 addresses in addition to IPv4 ones. Furthermore, as long as the transition lasts, a hodgepodge of IPv4 and IPv6 addresses can be returned for a single name, e.g.:

```
www.example.org.    IN    A       192.0.2.77
                          AAAA    2001:db8:c001::beef
```

Handling addresses from different families right will ultimately be up to the application developers. First, they need to realize that all addresses are no longer IPv4. Second, they should take into account the Default Address Selection rules from Section 6.6. And what we are going to take care of now is making sure that an application can have easy access to all addresses a DNS name maps to.

Between the DNS client available from a network library and the application, an extra abstraction layer is often found, to provide a uniform API to different name resolution systems such as DNS, NIS, WINS, HOSTS.TXT, and whatnot. For example, the one provided by the traditional *BSD Unix* API was *gethostbyname()*. From there that function has made it into *POSIX* and the *Windows Sockets* API. Unfortunately, *gethostbyname()* can't support multiple address families coexisting as it can return a list of addresses from a single family only. By default it will return IPv4 addresses only, the way to switch it to IPv6 being system-dependent.

For example, the *gethostbyname()* implementation popularized by the *BIND* resolver will start returning IPv6 addresses instead if the *RES_USE_INET6* option is selected. This option can be enabled globally using its text name `inet6` in the *resolv.conf* file or the *RES_OPTIONS* environment variable.

As it can be seen, new name resolution functions are required. However, it is an undeniable truth that changing an existing API can be even more troublesome and painful than altering a network protocol. To make up for the trouble, the new API ought to be as flexible and comprehensive as possible, so as not to be disturbed ever again. For this reason we can't simply bolt IPv6 support on the existing IPv4-centric API.

The new API must be able to handle any address families at once. From a programmer's point of view, this isn't really hard to do: Each address just needs to go up or down the API along with the mandatory attribute of its family. One way to implement this is to store the address inside a compound data structure with another field indicating the address family, e.g., as an integer code.

> This technique won't be new to those having experience with the *Berkeley Sockets* API.

This is exactly how the new function *getaddrinfo()* behaves [63, Section 6.1]. Any address family value it returns can be transparently passed to the *Berkeley Sockets* functions [63], so applications no more have to handle each address family separately. Now a well-designed application will require no changes to be able to handle a new IP version or even a different protocol stack with compatible semantics.

Will the new API be able to support the IPv6 scoped address architecture from Section 2.4? If a name resolution system other than DNS is in use and it allows somehow to specify the correct zone_id strings in addresses published, the *getaddrinfo()* function will be perfectly able to pass them up to the application because the encapsulating structure for an IPv6 address, *sockaddr_in6*, already contains a field for this info.

By contrast, if an address was learned from a *AAAA* record in the DNS, known is only its numeric value not qualified with a zone_id. Why it is so was discussed in Section 2.11. However, having no zone_id in the right-hand side of the *AAAA* record still doesn't prevent the address specified there from being non-global. For instance, if site-local addresses are employed by a company, they implicitly belong to a certain zone, namely, the corporate network of the company. On the one hand, it is OK for such addresses to appear in the company's private DNS zone. On the other hand, they will be meaningless if outside of the corporate network.

Leaking non-global addresses into the public DNS can have adverse consequences. Consider the following example. Suppose company *A* has published the following resource records in the DNS:

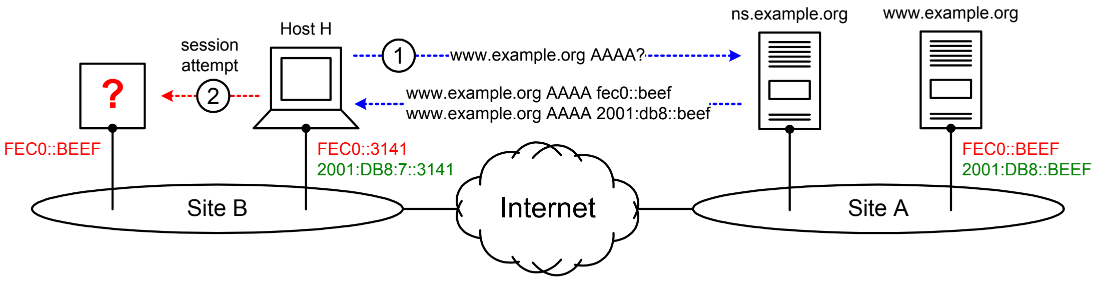


Figure 6.15: Careless publication of a site-local address in the public DNS

```
www.example.org.    IN    AAAA    2001:db8::beef
                          AAAA    fec0::beef
```

Company *B* is also using site-local addresses in its intranet. One of its hosts, *H*, wants to speak with *www.example.org*. As host *H* has a site-local address, Default Address Selection from Section 6.6 will prefer the remote address *FEC0::BEEF* because it has the smaller scope. However, in *H*'s home context, the zone of *FEC0::BEEF* is site *B*, not site *A*! When host *H* actually tries to send packets to that address, three basic outcomes will be possible, collectively shown in Fig. 6.15:

1. The communication will fail because there is no such address as *FEC0::BEEF* present in network *B*. This failure is non-fatal as Default Address Selection will proceed to try the next destination address available, which is the global address *2001:DB8::BEEF* suitable for this communication.

2. A session will be established with the owner of *FEC0::BEEF* in network *B*. Since it is a random host, not the real *www.example.org*, the result will be undefined.

3. An attacker who has already infiltrated network *B* will assume the unused address *FEC0::BEEF* and hijack the communication. This incident can be further exacerbated if host *H* applies different levels of trust to Internet and intranet hosts, e.g., makes authentication and encryption optional if speaking to a site-local peer.

The rules to avoid this fault will be as follows:

- *Non-global IPv6 addresses are not to appear in public DNS zones.*

- *A DNS client shouldn't return non-global addresses to the application if they are coming from a public DNS zone.*

The latter rule can be hard to implement in general because it is conditional on the client's ability to distinguish private and public DNS zones. However, when in a typical intranet, this task can be delegated to the recursive DNS server. When responding to stub resolvers' requests, it can be filtering out non-global records coming via other servers and letting its own authoritative answers through. To complete this scheme, the resolvers need to be prevented from speaking directly to external DNS servers, e.g., with a firewall policy.

6.8 Multihoming and Multipathing with IPv6

Caution! You are entering an area of research and experiment! This aspect of IPv6 operation is still under active development.

We are going to conclude our imaginary work on IPv6 by discussing the following seemingly trivial question: How can an IPv6 host make an effective use of multiple network connections? Until now, we have been laying the necessary grounds for that and, at the same time, going to great lengths to avoid facing the problem itself. Now the time has come.

To fully appreciate the value of this question, let's go, in our mind's eye, back to the times when IPv4 exclusively dominated the market, and see how a network would get a fault-tolerant connection to the Internet in those days.

As we know, the Internet is a network of networks.

The most popular recipe for that was, and still is, to employ redundant elements such as individual connections to peer networks so as to eliminate a single point of failure. So, simply speaking, the network needed more than one connection to a peer network, with such redundant connections preferably going to independent peers. However, for those connections to actually do their job, the network's addresses needed to be advertised into the global routing cloud of the Internet via each of the connections.

For application traffic to go one way, the routing information has to go the other way. This is an obvious feature of the IP routing architecture.

If the network was operated by a sufficiently large ISP having LIR status, an Autonomous System, and other essential attributes of a serious player, then the network would advertise its addresses via BGP and the rest would happen automagically. This is a well-established scheme for large networks, and IPv6 LIRs will also be advertising themselves via MBGP—a multiprotocol extension to BGP.

But what if the network belonged to some organization wanting to use the redundant Internet connections solely for its private purposes? Until IPv4 addresses became too scarce, such an organization could obtain an Autonomous System and a Provider-Independent (PI) block of addresses right from an RIR, then arrange connectivity to multiple ISPs and, by mutual agreement with each of them, start advertising itself via BGP over each connection. An alternative to that would be to get a block of addresses from one ISP and have the other ISPs advertise it too, subject to the first ISP's approval. Either way, the long-term effect would be ever greater fragmentation of the global address space and an increasing number of non-aggregatable routes in the default-free zone of the Internet. Indeed, each network would be represented with at least one route, and the total number of such networks in the world can exceed the number of ISPs by orders of magnitude. So it is unsurprising that PI assignments are disfavored by the current IPv6 address allocation policies as exercised by the RIRs.

The most "interesting" case would fall to networks that weren't big or important enough to get represented in the global routing cloud. They had no options but to get connected to several ISPs, receive a block of addresses from each of them, and then start building a system of dirty hacks for the network to actually use the multiple connections. Here is a brief and incomplete list of obstructions such a network would have to surmount:

- An ISP would rarely let a packet from its own customer through if its Source Address came from another ISP's address block. This was a reasonable measure against IP address spoofing.

- Inbound fault tolerance, such as required by a public web server, could be provided by assigning the server addresses from different ISPs and publishing them in the DNS. However, the server-to-client traffic would have to be routed via the right connection to avoid hitting the previous problem. Since the traditional IP routing process was insensitive to the Source Address, non-standard tricks such as policy routing had to be employed, negatively affecting the throughput.

- Outbound fault tolerance, for the network's own users to keep Internet connectivity while one of the connections was down, would be possible only through combining NAT with stopgap tricks to monitor the connection status, such as this: "Can't seem to ping Google via the

> current connection... The ISP must be down... Switching the default route over to the backup ISP..." A core problem here was that an IPv4 host would generally support just one address.

The extent to which these tricks could provide fault tolerance still was quite limited. In particular, it was impossible to preserve so basic a thing as a TCP connection because the local address of the socket would be forced to change when switching ISPs. All in all the smaller the network was, the harder the effective use of multiple Internet connections would become for it.

So now, at the time when great changes are possible, we should move our focus from large-scale mechanisms for fault tolerance, such as BGP, to something accessible to individual users. In other words, we need to make all networks happy, and the smallest network possible is just a single host. We therefore are going to zero in on **multihoming** solutions for an IPv6 host.

Suppose we are a host's owner. How will we ensure its Internet connectivity doesn't depend on a single ISP?

One option is to equip the host with two or more network interfaces and connect each of them to a different ISP—assuming the ISPs we have chosen are genuinely independent. Each interface thus will get an IPv6 address from its respective ISP. The reference IPv6 host is already capable of handling this kind of configuration because each of its interfaces has a separate Default Router List on it, as shown in Fig. 6.16.

> One can say that the reference IPv6 host supports network stack virtualization out of the box.

The other option is available when there are multiple ISPs present on one link; or multiple routers, each connected to a distinct ISP. In this case the host will need just one interface, but its interface will get multiple IPv6 addresses, one from each ISP. Then the host will have a single Default Router List with all of the ISPs represented in it, as illustrated in Fig. 6.17.

However, either multihoming scenario is just a first step to fault-tolerant connectivity. How can the host benefit from it? The host can directly detect a connected link failing or a default router going down thanks to implementing the NUD mechanism from Section 5.1. So the host will be able to fail over to another default router using only standard mechanisms. Not bad for a start!

At the same time, should a fault occur farther into the Internet, e.g., inside the current ISP's network or upstream of it, the host will get no signal about it. To see what will happen in this case, consider two scenarios, in which the host either accepts TCP connections from other hosts or originates such connections itself.

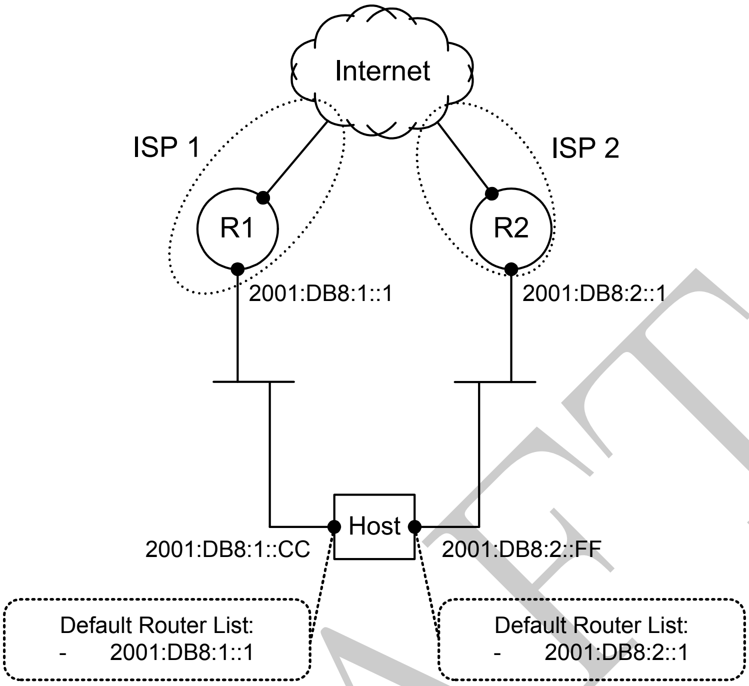


Figure 6.16: Multihoming with different links

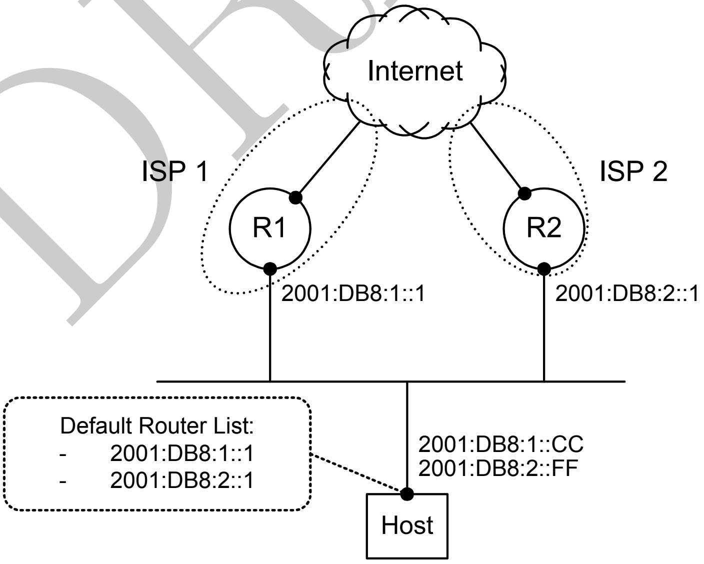


Figure 6.17: Multihoming with one link

Suppose that in the first scenario, where the host is to accept TCP connections, the host's addresses coming from different ISPs have been published in the DNS so the host's name resolves to all of them. Then a remote application implementing Default Address Selection from Section 6.6 will try the addresses one by one. Assuming the host's ISPs are independent, the network paths to its addresses will be different, and eventually the remote application will find a reachable address. So the opening SYN packet will come through, say, via ISP 2. Then our host will need to ensure that the SYN+ACK response is sent via the same ISP because it has proved to be up by delivering the initial SYN packet OK. This task boils down the selecting the right interface in Fig. 6.16 or Default Router List entry in Fig. 6.17. In the former case the solution can be as follows: the TCP socket needs to be bound to a specific interface. E.g., if the initial SYN came in via interface 2, the outgoing packets of this connection need to be sent via the same interface. As for the latter case, the host will need an additional routing rule for the packet, whose Source Address comes from ISP 2, to get sent via the default router provided by ISP 2. Thanks to the Destination Cache entry created by this routing decision, the subsequent packets of this connection will be sent via ISP 2 automatically.

Such a routing rule hasn't been made standard yet although it has been discussed [217][218].

If the same remote address establishes concurrent connections with our host's local addresses coming from different ISPs, the DC-based scheme will fail. To restore its usability, a DC entry will have to be sensitive to the source address so that different local addresses map to different DC entries even if the destination address looked up is the same.

Now let's move on to the second scenario, where our host is trying to open a TCP connection to a remote party by sending a suitable SYN packet. If the remote endpoint has multiple addresses and only some of those are unreachable via the current ISP, simply trying the remote addresses one by one will allow to find a reachable remote address. But what if the current ISP has silently and completely failed? The host will be able to detect such a failure only using some heuristics. A guideline for the host can be to try each of its interfaces in Fig. 6.16 or each Default Router List entry in Fig. 6.17. In the former case Default Address Selection will pick the right source address for the SYN packet. In the latter case the next hop chosen needs to influence the source address selection, which is an acceptable extension to the selection procedure [212, Sections 5 and 7].

An IPv6 host thus can make much use of multihoming through relying only on the standard mechanisms or most natural extensions to them. Nevertheless, we haven't provided for the most basic feature yet: to preserve an open TCP connection when switching ISPs. It is easy to verify that switching ISPs as discussed will make the socket change its local address anyway and the TCP connection will fail. Alas, it can hardly be helped at the IP layer.

The easiest way would be to dismiss this issue altogether by telling that application protocols must be able to cope with transport layer faults. No doubt, this is a very valid requirement. For example, the "reget" feature as supported by FTP and HTTP isn't sensitive to the local IP address; the client application will just open a new session and request the same data object at a particular offset.

Still, choosing to dismiss the problem would be boring and do no credit to our imagination. So let's try to rough out a more interesting solution, even if it will have to go beyond the IP layer. The raw idea is going to be pretty simple: Suppose the application can open a TCP connection with a whole set of remote addresses at once. Indeed, why to bother checking one remote address at a time when a SYN packet can be sent to each of them in parallel. By extension, a set of local addresses can be employed as well.

The naive way of emulating such a multi-address TCP connection would be to open $N \times M$ plain TCP connections, where N and M are the numbers of local and remote addresses, respectively. However, this approach would hardly be feasible because the number of connections so required can be large and they need to be additionally managed, at least to ensure byte ordering across individual connections. Indeed, the receiver must be able to recover the relative order of the bytes coming in via different connections.

So we need a more comprehensive solution instead. One option is to replace TCP with a new transport protocol designed with multihoming in mind. An IPv6 packet still has just one Source Address and one Destination Address, so handling multiple addresses on either endpoint will be up to the new protocol. In particular, it will have to continuously monitor pairwise address reachability and perform failover when necessary so that the application doesn't need to be concerned with all this fairly complex stuff. In fact, such a protocol has already been created: SCTP [126]. This solution's beauty is in that it is 100% end-to-end: the hosts involved don't care where the fault is localized, and as long as there is a mutually reachable address pair, the transport connection will live on.

However, to introduce a new transport protocol will be a quite radical innovation mostly suitable for new applications. Another issue with such a protocol will be that the various middlebox products such as firewall systems will take time to catch up with the advance and to start handling the new protocol no worse than the good old TCP. So let's next see if it can be possible in principle to retrofit the new ideas to TCP so that the existing apps and systems can benefit from multihoming.

The design of Multipath TCP (**MPTCP**) must be based on the principles of compatibility with the applications and the network [219]. What does it mean? On the one hand, the API such as the *Berkeley Sockets* must remain unchanged so that the existing applications can go without recompiling, let alone rewriting them. On the other hand, to an external observer each basic MPTCP path, known as a **subflow**, must appear as a normal TCP connection. The latter is required for subflows to pass through middleboxes unhindered.

> In particular, each subflow needs an independent continuous TCP Sequence and Acknowledgement numbering. If all subflows were to share the Sequence and Acknowledgement numbers, the protocol would be simplified because the original byte order could be recovered from the basic fields in the TCP header of an MPTCP segment; but a strict middlebox would block individual subflows for the apparent violation of the basic TCP spec by having gaps in the Sequence and Acknowledgement numbers. Mapping between the byte numbers in individual subflows and in the master connection is made possible thanks to the extra information carried by MPTCP options. Implemented as regular TCP options, they are expected to be transparent to middleboxes.

We have effectively returned to the idea of running multiple concurrent TCP connections between different pairs of local and remote addresses, but now they are to be controlled by a well-defined mechanism. To be efficient as well as effective, MPTCP will have to control not only reachability but also congestion in the subflows, moving bulk traffic away from congested ones [220]. The extra control information required by MPTCP can be placed in TCP options, which already see many uses and so should have no trouble passing through middleboxes [221]. Owing to the extra information, MPTCP can recover the original byte order from the subflows, and so it supports simultaneous data transmission over multiple paths for a greater throughput.

> Of course, MPTCP isn't specific to IPv6. What's more, it has no problem using both IPv4 and IPv6 paths in one connection.

And what about other protocols operating on top of IPv6? For all of them to benefit from multihoming, a different solution can be proposed. Suppose a “shim” layer is inserted between such a protocol and IPv6 to manipulate the addresses in the packets so that the upper-layer protocol always sees just a pair of addresses, local and remote, as before. This extra layer, known as **Shim6** [222][223], will have to perform dynamic translation of the constant “upper” addresses to the “lower” addresses dependent on the current reachability situation in the network and vice versa. This translation turns the addresses visible to the upper-layer protocol into identifiers independent of the current network topology while the addresses to be seen on the wire become locators sensitive to what is happening in the network. For this role the former are known as **Upper-Layer Identifiers** (**ULID**). Of course, a ULID remains a regular IPv6 address and so it can also be a locator as long as its network interface is reachable. Effectively an IPv6 extension, Shim6 conveys its control information in IPv6 extension headers dedicated to the purpose. The information about locator reachability is supplied to Shim6 by an auxiliary protocol, **REAP** (Reachability Protocol) [224].

Discuss whether Shim6 is going to cause the same problems as NAT to application protocols that pass IP addresses in their messages—think FTP and SIP.

We won’t delve into the fine details of MPTCP or Shim6 because they are just too complex to fit in our course nicely. Still, the bits we have already discussed are enough to see that the end-to-end model has won yet again, enabling a host to make effective use of multihoming with no help from intermediate nodes; and IPv6 has provided the host will all tools required.

Needless to say, the use of multiple paths and addresses to send and receive data entails new problems. One of them is how to ensure that all remote addresses learned actually belong to the same remote host. Otherwise an attacker would be able to hijack the communication by inserting its own address into the remote address set. This is of particular relevance to protocols such as Shim6 in which an endpoint isn’t required to indicate all of its addresses at the beginning since they can be added to the session as it goes.

To bind multiple addresses together, their interface identifiers can be used again to carry the security information. Suppose N addresses are to be assigned to a host, all of them coming from different subnets. So given is a list of subnet prefixes $\{P_i\}$, where $1 \leq i \leq N$. Then the interface ID L_j for the prefix P_j can be calculated as a hash value of *all* prefixes, e.g., as follows:

$$L_j = H\left(M|P_j|\{P_i\}\right).$$

That is, the hash function is computed over the bit string obtained by the concatenation of a random modifier value M known to both parties, the subnet prefix P_j to be used in this address, and the list of all prefixes $\{P_i\}$. So after appending the corresponding interface ID to each subnet prefix, we are going to get a set of N such addresses:

$$A_j = P_j | L_j.$$

Each of the addresses so built is known as a **Hash-Based Address** (**HBA**) [225]. It retains its "magic" properties only as long as in the **HBA-Set** $\{A_i\}$; when alone it is just an ordinary IPv6 address.

The modifier M is needed, for one thing, to allow different hosts connected to the same subnets to generate unique HBA-Sets. Besides, an HBA is unpredictable due to the use of a hash function, and so a conflict with an existing address can't be ruled out. The conflict, should it occur, will be detected by the DAD procedure from Section 5.4.1. But to actually resolve the conflict, the modifier will need to be varied and the HBA calculation, redone. In reality, such a modifier is composite and consists of several fields, each responsible for one HBA-Set feature such as uniqueness, conflict tolerance, compatibility with CGA from Section 6.3.

Providing the hash function is strong enough, the order in which its input is pieced together is insignificant as long as it is the same in all protocol parties. In practice, one of the modifier fields, namely, the collision count, comes between the prefix P_j and the list $\{P_i\}$ for the sake of compatibility with CGA. This field is incremented each time DAD detects a conflict with an existing address. Thanks to that, redoing the calculation after a conflict will automatically yield a different HBA-Set.

Now consider the other endpoint. How can it validate some address A its remote peer is claiming? Let the address consist of subnet prefix P and interface ID L. The validation procedure will boil down to making sure that the given address A indeed belongs to the HBA-Set of the remote peer. To perform it, the validator will need to know beforehand the modifier M and the prefix list $\{P_i\}$. It is these two parameters that define the HBA-Set because the rest can be computed on demand. First of all, the validator needs to check that the prefix P is actually present in the prefix list $\{P_i\}$. If it isn't, the address A is obviously rogue, corrupted, or just wrong due to a fault or bug. And if this simple test has been passed OK, next is math time. Namely, the expected HBA interface ID is to be computed:

$$L_{HBA} = H\left(M | P | \{P_i\}\right).$$

If the address A genuinely comes from this HBA-Set, its interface ID L must be the same as L_{HBA} except for the **U/L** and **I/G** bits.

The last detail to take care of is how the required information consisting of the modifier M and the prefix list $\{P_i\}$ is to be conveyed to the HBA validator. As it happens, no new protocol is needed for that because the information can be passed in a CGA Parameters structure from Section 6.3. The CGA Parameters format is flexible enough to provide extra room for the prefix list while a field for the modifier is already in place. This allows one to build an IPv6 address that will be a proper CGA and a valid HBA at the same time thanks to its interface ID being a standard function of both the public key and the prefix list. So a potential conflict between the semantics of CGA and HBA is miraculously avoided.

The CGA Parameters format (Fig. 6.3 in page 289) already contains a random modifier, the subnet prefix P, and a collision count; together these pieces constitute the $M|P$ substring in the HBA hash function input. The prefix list $\{P_i\}$ can be stored at the end of the structure, as its extension. Thus the HBA hash function input will be formatted as a CGA Parameters structure, making HBA just a compatible extension of CGA.

Unsurprisingly, an IPv6 packet using a plain HBA won't pass CGA validation because it isn't really signed with a private key. Instead, HBA-without-CGA puts a random bit string conforming with the RSA key format in place of a real public key. Still, the HBA-only address will contain a mathematically valid fingerprint of this random key.

Being a compatible extension to CGA, HBA can't but inherit its restriction on the subnet prefix length, which has to be 64 bits as long as either of these mechanisms is in use.

Conclusion

Now that our first tour through the IPv6 realm is finally over, a brief summary is due. First of all, we should point out that not all important details of the IPv6 technology have made it into the course. In particular, we deliberately omitted such practical aspects as design, configuration, and troubleshooting of IPv6 networks. Quite a few vendor-specific textbooks have already been published on those topics while our goal was to provide a solid basis of knowledge underneath so that the reader themself could solve essentially new problems with no quick answers available from the textbooks, as they are bound to be encountered when working with IPv6.

Besides, the mechanisms for gradual and, ideally, painless transition between IPv4 and IPv6 have been completely ignored for the time being. Granted, the Transition is a big and fascinating topic but, for its study to go well and be fruitful, the plain vanilla IPv6 first needs to be grasped in full. Hoping our little team with the reader on board has proved equal to the primary task, our dear teammate should be fit to continue along this exciting path on their own now. Farewell!

As for us the authors, we would like to close with re-emphasizing the strength of IPv6 we appreciate the most. We believe that the greatest merit of the IPv6 architecture is in that it restores the end-to-end transparency of the Internet and puts in a proper context such diverse problems as addressing, security, and privacy [226]. This isn't only important because an attempt to control those problems with the silver bullet of NAT has led to ineffective and unreliable solutions. Alas, the way things are being done in the IPv4 industry today encourages a sloppy manner of technical reasoning when an engineer, instead of conducting thorough analysis and finding a structural approach to the project (see Fig. 6.18), either cobbles up some combination of units he never bothers to understand but believes to "usually work" or, even worse, just offers his favorite one-size-fits-all solution. The management for their part are eager to make that junk their flagship product because it is fast, cheap, and requires no competent staff. It is no surprise this situation discourages engineers from ever learning to think on their own and

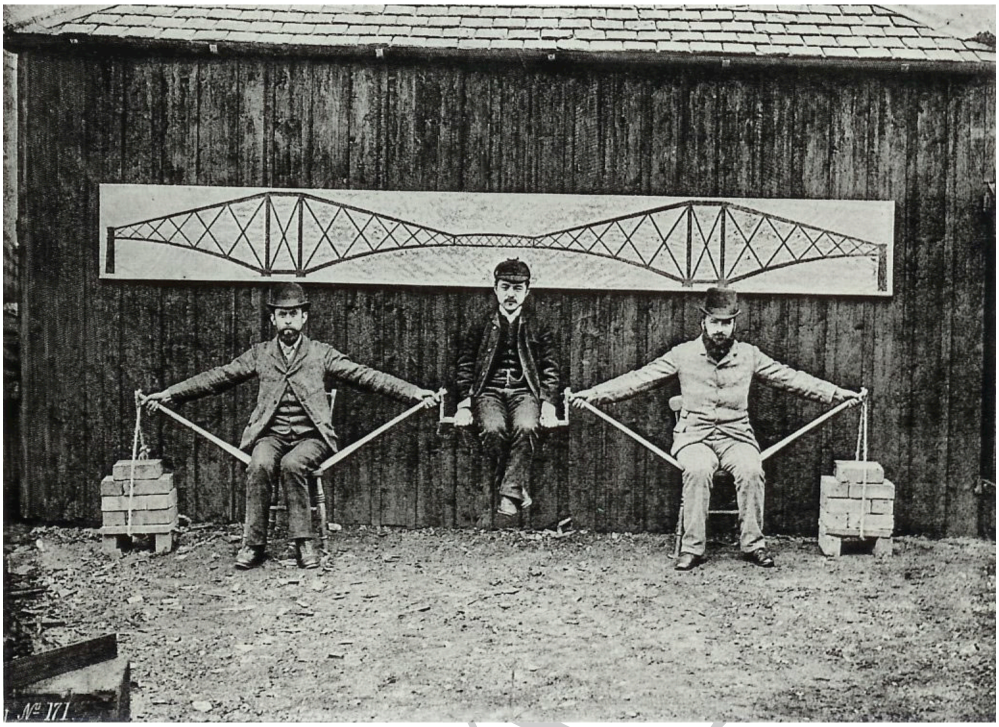


Figure 6.18: Engineers B. Baker, K. Watanabe, and J. Fowler demonstrate the structural principles of the cantilever bridge [227]. Their designs can be relied upon!

hardly helps them to enjoy their jobs, much less to become true experts. So the advent of IPv6 comes as the opportunity of a lifetime for the TCP/IP community to restore the persistent intellectual process thanks to which the unique technical phenomenon of the Internet was born in the first place.

Bibliography

[1] IPv4 Exhaustion. RIPE.

[2] RFC 5735

[3] RFC 3021

[4] RFC 3194

[5] RFC 4692

[6] G. Huston. IPv4 Address Report.

[7] RFC 2775

[8] U. Rivner. Anatomy of an Attack. RSA.

[9] L. Phifer. The Trouble with NAT. The Internet Protocol Journal - Volume 3, No. 4.

[10] The University of Michigan Information Technology Security Services. Security Considerations Of NAT

[11] RFC 3715

[12] RFC 3424

[13] RFC 2993

[14] Version Numbers. IANA registry.

[15] RFC 1347

[16] RFC 1475

[17] Archimedes. The Sand Reckoner.

[18] P. Smith. BGP Routing Table Analysis. APNIC.

[19] RFC 5375

[20] Conservation. IPv6 Address Allocation and Assignment Policy. RIPE.

[21] RFC 4291

[22] RFC 4903

[23] Guidelines for 64-bit Global Identifier (EUI-64$^{\mathrm{TM}}$) Registration Authority. IEEE.

[24] IPv6 Address Allocation and Assignment Policy. RIPE.

[25] RFC 1219

[26] RFC 3531

[27] RFC 3986

[28] RFC 20

[29] RFC 5198

[30] RFC 1924

[31] IEEE 802-2001

[32] RFC 822

[33] RFC 4632

[34] RFC 5952

[35] RFC 3927

[36] RFC 3879

[37] RFC 3596

[38] RFC 1035

[39] RFC 4038

[40] RFC 3587

[41] Internet Protocol Version 6 Address Space. IANA registry.

[42] IPv6 Global Unicast Address Assignments. IANA registry.

[43] RFC 5737

[44] RFC 3849

[45] RFC 5156

[46] RFC 4007

[47] ID draft-fenner-literal-zone

[48] ID draft-ietf-6man-uri-zoneid

[49] Andrew S. Tanenbaum. Computer networks, Fourth Edition.

[50] RFC 4862

[51] RFC 1122

[52] RFC 5340

[53] RFC 6164

[54] Guidelines for use of the 24-bit Organizationally Unique Identifiers (OUI). IEEE.

[55] RFC 3513 EUI-48/MAC-48 confusion (IETF IPng WG mailing list thread).

[56] why 0xFFFE is used in the modified EUI-64 format (IETF IPv6 WG mailing list thread).

[57] IPv6 Multicast Address Space Registry. IANA registry.

[58] IPv4 Multicast Address Space Registry. IANA registry.

[59] RFC 3956

[60] RFC 2365

[61] RFC 6676

[62] RFC 3092

[63] RFC 3493

[64] RFC 1112

[65] RFC 2730

[66] RFC 4489

[67] RFC 3306

[68] RFC 6034

[69] RFC 4607

[70] RFC 3569

[71] RFC 1958

[72] RFC 6308

[73] RFC 3307

[74] M. Sayrafiezadeh. The Birthday Problem Revisited. Math. Mag. 67, 220-223, 1994.

[75] Accident statistics. Plane Crash Info.com.

[76] RFC 4193

[77] ID draft-hain-ipv6-ulac

[78] RFC 6177

[79] RFC 1918

[80] RFC 5453

[81] inet(3). FreeBSD manual pages.

[82] RFC 2460

[83] Protocol Numbers. IANA registry.

[84] RFC 1812

[85] RFC 791

[86] The KAME project.

[87] RFC 2419

[88] RFC 2675

[89] RFC 2474

[90] RFC 3168

[91] RFC 6437

[92] RFC 6438

[93] ID draft-carpenter-v6ops-label-balance

[94] RFC 793

[95] RFC 4302

[96] Internet Protocol Version 6 (IPv6) Parameters. IANA registry.

[97] JINMEI Tatuya. Re: IPv6 Scoped Address Architecture (RFC 2710). IETF IPng WG mailing list.

[98] J. Postel, J. Reynolds. Comments on the IP Source Route Option. ISI, 1987.

[99] P. Biondi, A. Ebalard, "IPv6 Routing Header Security", CanSecWest Security Conference 2007, April 2007.

[100] RFC 5095

[101] N. Guilbaud. Google Backbone monitoring. Localizing packet loss in a large complex network. AusNOG 2012.

[102] RFC 3775

[103] RFC 1191

[104] RFC 4919

[105] RFC 4944

[106] RFC 4963

[107] RFC 5722

[108] RFC 6434

[109] RFC 4294

[110] RFC 4303

[111] M. Kaeo "IPsec Analysis in an IPv6 Context–v01", NAv6TF, 2007. (alternative archive link)

[112] RFC 4301

[113] RFC 3948

[114] ID draft-krishnan-ipv6-exthdr

[115] RFC 1055

[116] RFC 894

[117] Steve Deering. Why a new etype of IPv6. Internet-history mailing list.

[118] Radia Perlman. Interconnections: Bridges, Routers, Switches, and Internetworking protocols (2nd Edition).

[119] Radia Perlman. Mythology and Folklore of Network Protocol Design. 2005.

[120] Radia Perlman. Mythology and Folklore of Computer Network Protocol Design. 2008.

[121] RFC 2464

[122] RFC 5342

[123] RFC 1332

[124] RFC 5072

[125] Point-to-Point (PPP) Protocol Field Assignments. IANA registry.

[126] R. Stewart, M. Tüxen, P. Lei. SCTP: What is it, and how to use it? BSDCan 2008.

[127] IEN-45

[128] RFC 1071

[129] RFC 1624

[130] RFC 2473

[131] RFC 3056

[132] RFC 4213

[133] RFC 4459

[134] RFC 792

[135] RFC 4443

[136] Internet Control Message Protocol version 6 (ICMPv6) Parameters. IANA registry.

[137] Gary R. Wright, W. Richard Stevens. TCP/IP Illustrated, Volume 2: The Implementation.

[138] RFC 4620

[139] RFC 4861

[140] Qing Li, Tatuya Jinmei, Keiichi Shima. IPv6 Core Protocols Implementation. Morgan Kaufmann, 2007.

[141] IEEE 802.1ak-2007

[142] RFC 4541

[143] RFC 5082

[144] RFC 826

[145] RFC 2991

[146] S. Floyd, V. Jacobson. The Synchronization of Periodic Routing Messages. IEEE/ACM Transactions on Networking, April 1994.

[147] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, Marc Bechler. The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel. Prentice Hall, 2004.

[148] Introduction to Mobility in IP. Configuring Proxy Mobile IP. Cisco Aironet 1200 Series Access Point Software Configuration Guide for VxWorks.

[149] Description of Address Resolution Protocol (ARP) caching behavior in Windows Vista TCP/IP implementations. Microsoft KB article 949589.

[150] RFC 5681

[151] RFC 5942

[152] RFC 5308

[153] RFC 4311

[154] ISO 9542:1988. Information processing systems – Telecommunications and information exchange between systems – End system to Intermediate system routeing exchange protocol for use in conjunction with the Protocol for providing the connectionless-mode network service (ISO 8473).

[155] RFC 995

[156] RFC 4943

[157] Using IP Multicast Over Frame Relay Networks. Cisco technology white paper.

[158] RFC 5966

[159] RFC 1546

[160] RFC 3258

[161] RFC 2526

[162] ifconfig(8). FreeBSD manual pages.

[163] Anycast IPv6 addresses. Microsoft TechNet.

[164] RFC 5227

[165] why is NA to DAD NS sent to all-nodes address? (IETF IPv6 WG mailing list thread).

[166] RFC 3315

[167] RFC 1256

[168] RFC 4429

[169] RFC 4191

[170] RFC 4086

[171] RFC 1201

[172] RFC 2497

[173] RFC 4192

[174] RFC 5969

[175] RFC 3627

[176] RFC 4941

[177] RFC 4135

[178] RFC 3756

[179] ID draft-arkko-manual-icmpv6-sas

[180] ID draft-arkko-icmpv6-ike-effects

[181] RFC 3972

[182] T. Aura, M. Roe. Strengthening Short Hash Values.

[183] RFC 4270

[184] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone. Handbook of Applied Cryptography.

[185] RFC 4982

[186] RFC 3971

[187] ID draft-krishnan-cgaext-proxy-send

[188] RFC 6104

[189] RFC 988

[190] RFC 3590

[191] RFC 3810

[192] RFC 2711

[193] IPv6 Router Alert Option Values. IANA Registry.

[194] RFC 2710

[195] RFC 2236

[196] RFC 3376

[197] RFC 4286

[198] RFC 4604

[199] RFC 5790

[200] `http://www.xorp.org/`

[201] Marc “vanHauser” Heuse. Recent advances in IPv6 insecurities. 27th Chaos Communication Congress. Berlin, 2010.

[202] RFC 3740

[203] RFC 2923

[204] RFC 4821

[205] RFC 1981

[206] RFC 3542

[207] ID draft-park-pmtu-ipv6-option-header

[208] RFC 1661

[209] RFC 6691

[210] RFC 1323

[211] RFC 4177

[212] RFC 6724

[213] RFC 1123

[214] RFC 5014

[215] RFC 5220

[216] RFC 5221

[217] ID draft-huitema-multi6-hosts

[218] Kenji Ohira. IPv6 Address Assignment and Route Selection for End-to-End Multihoming. IETF Meeting 57.

[219] RFC 6182

[220] RFC 6356

[221] ID draft-ietf-mptcp-multiaddressed

[222] RFC 5533

[223] A. García-Martínez, M. Bagnulo, I. van Beijnum. The Shim6 architecture for IPv6 multihoming. IEEE Communications Magazine, 2010, v. 48, issue 9, pp. 152–157.

[224] RFC 5534

[225] RFC 5535

[226] RFC 4864

[227] `http://en.wikipedia.org/wiki/File:Cantilever_bridge_human_model.jpg`