IPv6/IPv4 Protocol Translation in SPIN

Vincent K. Lam
vkl@cs.washington.edu
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

1 Introduction

To ease the transition of the Internet from IPv4 to IPv6, header translation has been proposed to provide a truly incremental upgrade path. Translating between IPv4 and IPv6 involves two distinct procedures, address translation and protocol translation. The protocol translation procedure indeed depends on the type of address used, but its core concepts and associated issues are generally independent of any address translation mechanism. This paper focuses on the protocol translation aspect and details it using a translator implementation done in the SPIN extensible operating system [1]. Although this specific implementation is discussed, the protocol translation procedure described is fundamental to all IPv6/IPv4 translators.

Marc Fiuczynski, a graduate student at the University of Washington, and myself have implemented a Network Address and Protocol Translator (NAPT) [2], in the context of SPIN, that performs the basic translation of packets from IPv4 to IPv6 and vice versa. For example, the translator can correctly translate cross protocol accesses of WWW, FTP, and Telnet services.

Section 2 of the paper gives an overview of the NAPT translation implementation. Details of the core protocol translation mechanism and accompanying issues are described in Section 3 and Section 4 respectively. Finally, conclusions are presented in Section 5.

2 Implementation

In order to implement the NAPT in SPIN, we had to start by adding IPv6 support to the operating system. Written under the Plexus networking architecture [3], SPIN already had support for IPv4 over Ethernet.

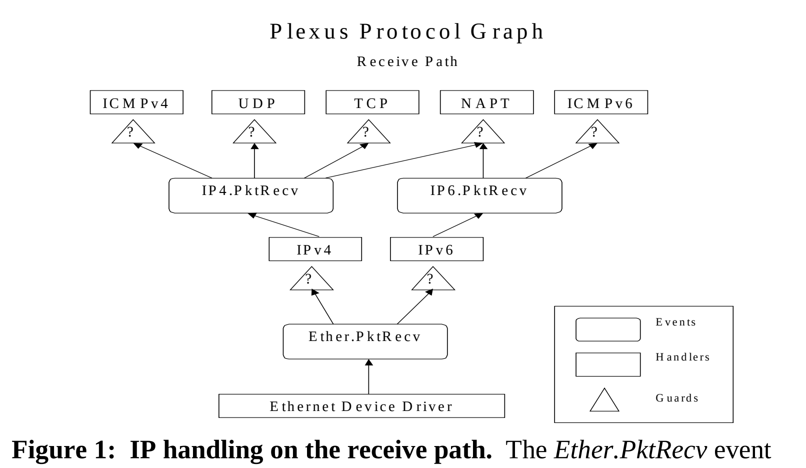

Figure 1: IP handling on the receive path. The *Ether.PktRecv* event causes either the IPv4 or IPv6 handler to fire. These handlers then inspect the IP header to determine whether to invoke the translating NAPT handler or just an upper-layer protocol handler.

Plexus allows for protocols to be dynamically added or modified in the kernel-level protocol graph through user-level extensions. Therefore we were able to add IPv6 support (as well as the translator functionality) without having to touch the SPIN kernel. Under the event-driven packet handling of Plexus, processing of a packet is easily modified by adding event guard and handler pairs. For example, adding handling for incoming IPv6 packets simply entailed attaching an associated guard and handler on the *Ether.PacketRecv* event. The guard inspects the Ethernet *protocol* field to determine whether the IPv6 handler should proceed. Figure 1 illustrates the IP handling currently implemented for the receive path. When a packet arrives from the Ethernet device, it is filtered using the IP guards. The NAPT guard then checks if the packet IP destination address is within the pre-defined block used for packets to be translated. If so, the NAPT handler is invoked and makes a procedure call to the appropriate translation function, as determined by the IP version of the packet. Note that handlers for TCP and UDP are not necessary in the case of IPv6 packets, since a router implementation of IPv6 (as opposed to a host implementation) is sufficient for the translator.

Figure 2 gives an overview of the packet translation procedure of the NAPT handler. Although this is actually implemented in the NAPT handler as two separate procedures, IPv4 to IPv6 and vice versa, the

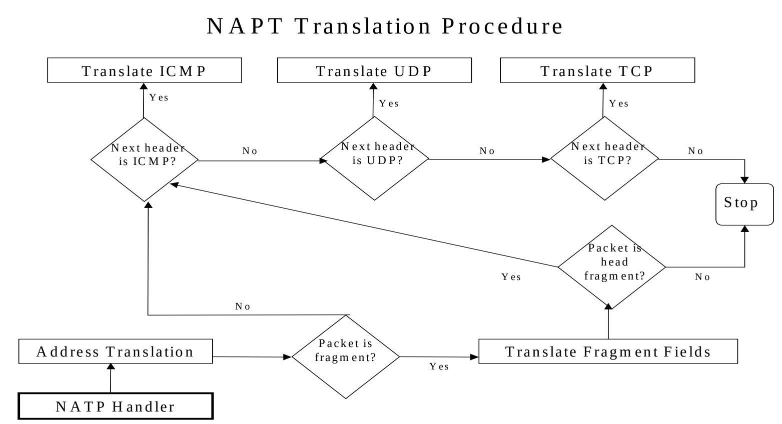

Figure 2: Basic packet translation procedure. After having its addresses translated, each layer of a packet is translated. Only the first fragment of a fragmented packet needs translation.

procedural flow is the same in both. The new IP addresses are determined during address translation and then each layer of the original packet is translated (in the manner detailed in Section 3). Mbufs are passed between the translating procedures and the resulting mbuf is then just sent to a procedure that emits IPv6 packets.

Currently, the IPv6 layer is a lightweight implementation that incorporates only the set of services necessary to support the translator (i.e. it does not completely implement the full router requirements as outlined in the IPv6 specification [4]). The IPv6 layer can respond to ICMP Echo Requests and Neighbor Solicitations (for link-layer address resolution) and supports fragmentation of outgoing packets. However, it currently does not perform any route lookups, do path MTU discovery (not required in IPv6, but recommended), maintain a neighbor cache, or respond to Router Solicitations (useful for auto-configuration of nodes in the IPv6 cloud).

Module	**Source Size (lines)**
NAPT	1685
NAPT protocol translation only	1630
IPv6	647
IPv4	1020
IPv4 with route	1471

lookup	
ICMPv6	723
ICMPv4	651

Table 1 lists some size metrics for the modules involved in the entire translation process. These modules were written in Modula-3 and compiled for the DEC Alpha. Note that the IPv6 layer is only about 650 lines of code. However, as mentioned above, it currently does not implement such features as route lookup. Assuming that the increase in size with these features implemented will be comparable to that of IPv4, the improved IPv6 layer would be about 1100 lines of code. The table also shows that the source size of the ICMPv6 layer is larger than that of ICMPv4. This was expected since ICMPv6 has more requirements than ICMPv4 (e.g. Neighbor Discovery). The minimal nature of the size increase is due to the fact that our ICMPv6 implementation only includes essential services and does not fully incorporate the full requirements outlined in the ICMPv6 RFC [5].

3 Protocol Translation Mechanism

The basic operation in protocol translation is to replace the IP header of the original packet with a new header of a different IP version. Whether upper-layer protocol headers get similarly replaced instead of just modified depends on the degree of difference between the protocol versions.

This section will discuss the protocol translation component of the overall packet translation procedure illustrated in Figure 2. Which fields to translate and how they should be updated will be discussed for each pertinent protocol, starting with the IP layer.

3.1 IP translation

Translating the core fields in the IP header is a straight-forward operation since many of the fields directly map from one protocol to the other. We further simplified the procedure by choosing not to translate extensions or options, except in the essential case of IPv6 fragments.

Figure 3 shows the translation of the core header fields. Most of the fields conveniently translate between protocols. For example, when translating from IPv4 to IPv6, the *ttl* field is directly copied to the resulting packet's *hop limit* field and decremented (to prevent routing loops between translators). For ease of implementation, the length fields are re-computed from the entire translated packet instead of being adjusted based on upper layer translations. In the case of stateless translation, the address fields are simply converted to v4-mapped and v4-compatible IPv6 addresses in accordance to [6]. When stateful translation [2] is used, the address fields are set according some address mapping maintained by the NAPT.

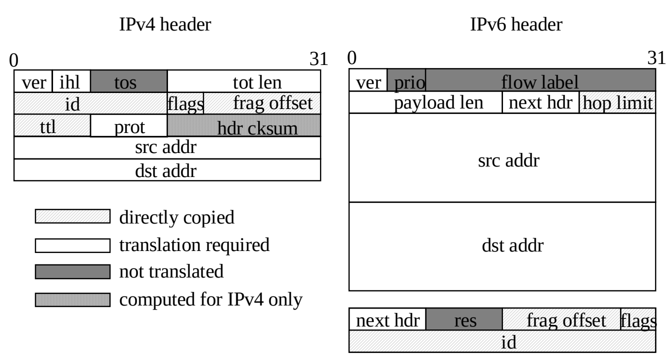

Figure 3: IPv6/IPv4 header fields. Many of the fields require translation, but fortunately all such fields conveniently map from one protocol to the other. Shown below the IPv6 base header is the IPv6 fragment extension header.

Unlike the fields mentioned above, the quality of service or flow related fields do not have the same translation convenience. IPv4 *type-of-service* values cannot be equivalently expressed in an IPv6 context where quality of service for a packet is marked with different semantics by the *prio* and *flow label* fields. The translator simply sets these fields to some default static values.

3.1.1 Fragmentation

When translating from IPv4 to IPv6, we increase the packet size by at least 20 bytes due to the header length difference between the two protocols. Since IPv6 routers cannot fragment, we can't send translated packets larger than the path MTU size. However, doing path MTU discovery is prohibitively expensive for a device meant to be transparent. Thus we resort to limiting the translator to sending IPv6 packets no larger than the IPv6 min MTU size of 1280 bytes. So if the *Don't Fragment* bit of the IPv4 packet is not set and the resulting translated packet is greater than the min MTU, then the translator will try to fragment bimodally into min MTU-sized packets.

Note that this will result in an inefficient packet stream in the case where the IPv4 host is sending actual MTU-sized packets to try to maximize throughput (i.e. an NFS client or server). For this situation, we are experimenting with returning to the IPv4 host an ICMP "Need to Fragment" error message that contains a "preferred MTU" size, giving the host the opportunity to re-adjust any path MTU value it keeps. This preferred MTU is equivalent to the min MTU minus 20 bytes for the header difference and 8 bytes for a possible fragment extension header. If the host continues to send large packets after receiving the error message, then the translator will stop sending the message and begin fragmenting. A drawback of this smarter fragmenting method is the slightly more complicated implementation and greater memory requirement to keep the notification state of the IPv4 hosts.

Since translating from IPv6 to IPv4 always results in a smaller packet size and IPv4 routers can indeed fragment, dealing with large translated packets in this direction is much easier than in the other. We simply fragment the translated packet bimodally if it is larger than the IPv4 next-hop MTU.

In both cases above, we choose to fragment large packets bimodally because we have observed that most Internet traffic is also fragmented as such. Therefore many routers and hosts are optimized to handle such packet streams.

3.1.2 Translating fragments

Not only does the translator must be able to generate fragments, it must be able to translate them as well. However, translating fragments is convenient because of the semantic equivalence of the fragment fields in the IPv4 header and the IPv6 fragment extension header (as shown in Figure 3). The *id* and *frag offset* fields and *More Fragments* flag are directly copied, with a special case existing when translating IPv6 fragments to IPv4 since the IPv6 fragment *id* field is twice as large as its 16-bit IPv4 counterpart. In this case, we simply copy the lower 16 bits of the IPv6 *id* field. Because the *id* field is typically generated by increments of one, the probability of duplicate IDs should be equivalent to that of a normal IPv4 session.

3.2 ICMP translation

Since ICMPv4 [7] and ICMPv6 [5] share the same header (*type*, *code*, and *checksum* fields) and most of the message types have semantic equivalence across versions, translation of ICMP is mostly straight-forward with just a few special cases.

One of these cases is the *checksum* field. ICMPv6 requires that its checksum include an overlay header (equivalent to the one used by TCP and UDP), whereas ICMPv4's checksum only covers the ICMP header itself. Therefore these checksums must be adjusted during translation. For ease of implementation, we simply re-compute this checksum after the ICMP header is fully translated. Trying to optimize this operation by keeping track of changes and taking checksum differences would be non-trivial, especially in the case of recursive ICMP translations (explained later in this subsection).

The only ICMP Query Messages that are translated are Echo Request and Echo Reply messages. The rest are silently dropped because they are either obsolete or are single-hop only. Since both versions of ICMP share Echo *identifie*r, *sequence number*, and *data* fields, we translate such messages by simply copying the original ICMP packet and then updating the *type* and *checksum* fields.

For the same reasons as for ICMP Query Messages, the only ICMP Error Messages that are translated are *Destination Unreachable*, *Time Exceeded*, *Parameter Problem*, and *Packet Too Big* (IPv6 originated).

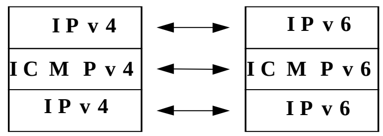

Figure 4: Recursive translation of IP header in ICMP Error Message.

Again, because of the equivalence of most of the ICMP messages, translating between versions mainly involves doing static conversions of the field values. The exact field translation details are discussed in [6]. However, one important detail to note is that ICMP Error Messages contains as much of the error invoking packet's IP header and payload as can fit, and thus that needs to be translated as well. A recursive translation of that IP information (illustrated in Figure 4) is necessary, which will likely change the size of the overall translated packet.

3.3 TCP and UDP translation

When translating TCP and UDP packets, the only field in the transport layer we need to give consideration to is the *checksum* field, since it is based on an overlay header that contains IP addresses.

In the case of stateless translation, we are fortunate to not have to modify this checksum because of the use of v4-mapped and v4-compatible addresses. Since these *special* addresses have a 96-bit prefix containing only zeros or ones, their one's complement checksum is equivalent to the checksum of their lower 32 bits. Thus translation of a TCP or UDP packet that uses these special addresses does not change the checksum.

Because stateful translation make use of all valid IPv6 addresses, the same convenience does not apply and the checksum must indeed be updated. However, instead of completely re-computing the checksum, we improve performance by just taking a checksum difference. Ignoring the checksum itself, the only field values in the TCP or UDP checksum calculation that are different between IPv4 and IPv6 packets are the address fields of the overlay header. Thus, as shown in Figure 5, the new transport layer checksum can be derived by just doing a ones complement sum of the address checksum difference and the original checksum.

new_cksum = orig_cksum + (cksum(new_src_addr, new_dst_addr) - cksum(orig_src_addr, orig_dst_addr))

Figure 5: Transport layer checksum calculation.

4 Issues

The above section shows that the core mechanism for translating from IPv4 to IPv6 is relatively straight-forward given some workarounds to handle cases like fragmentation and ICMP. However, there are some significant issues that may affect the overall feasibility of this translation. Loss of information and embedded IP addresses are two such issues and are addressed in this section.

4.1 Loss of information

In order to perform the basic header translation, essential fields containing version, length, protocol type, hop limit, and address information map directly from IPv4 to IPv6 and vice versa. However there are other fields that don't have semantic equivalence from one protocol to the other, resulting in a loss of information after the header translation. As discussed in section 2.1, the quality of service fields are examples of such. Another example is the use of extension headers by IPv6. These headers can be of arbitrary length and can encapsulate options greater than the IPv4 limit of 40 bytes. The IPv6 specification also defines extensions that are a superset of the IPv4 feature domain. Thus it is not possible for fully transparent header translation to occur without loss of information in cases where the disjoint functionality is exploited. One could imagine that this could be emulated by some sort of encoding scheme that would require a decoder at the other end of the transmission. But clearly this would destroy the desired transparency of the header manipulation.

It turns out, however, that most applications currently do not use the extended fields of the IP header. For example the application or presentation layer usually handles security. In practice, most applications do not depend on the IP layer for more than basic packet routing and switching.

4.2 IP address content in application-layer protocols

As discussed in Section 3.3, TCP and UDP are examples of upper-layer protocols that the translator needs to be aware of. Generally, the translator must know about upper-layer protocols that store IP address information and make appropriate updates so that the entire packet is accurately translated. However, there are many application-layer protocols, like FTP, that have this dependence and it is not feasible for the translator to accommodate all of them. The trade-off in increased complexity and time to perform these specialized header manipulations dictate that only a few such cases should be handled by the translator.

The only application-layer protocol that our translator supports is the FTP protocol, since it comprises a major portion of current Internet traffic. An FTP client embeds its IP address in the PORT command that it often sends to an FTP daemon. Because this address is stored in ASCII, the translator cannot update this field between the IPv4 and IPv6 address formats without changing the header size and updating TCP ack and sequence numbers to correspond. This method is quite complex, but is the only option when using stateful translation.

Fortunately, the use of the special IPv4-mapped and IPv4-compatible addresses in stateless translation allows FTP sessions to work without translation of the FTP header; only the IPv6 FTP applications need to be aware of the cross protocol access. When an IPv6 FTP client is used to connect to an IPv4-mapped address, it knows that the target FTP daemon speaks only IPv4 and thus sends a PORT command that contains the lower 32 bits of its IPv4-compatible address. When an IPv4 client connects to an IPv6 daemon, the daemon recognizes that the sender is an IPv4 host by the IPv4-mapped source address in the IP header. The daemon then knows to treat the 32-bit IP address in the PORT command as if it was an IPv4-mapped IPv6 address, thus enabling communication with the IPv4 client without any application-layer translation. This method of relying on the applications themselves to handle cross protocol accesses is the most low-impact way to support application-layer protocols that embed IP address information.

5 Conclusions

This paper has detailed the fundamentals of protocol translation as well as the significant issues involved in performing such a task. While the protocol translation was described in the context of a specific implementation, the concepts apply to all IPv6/IPv4 translators.

Leveraging the extensibility of the SPIN operating system and its Plexus networking architecture, we have implemented a fully functional Network Address and Protocol Translator capable of translating the protocols that comprise the majority of current Internet traffic. For example, the NAPT can translate cross protocol use of WWW, FTP, and Telnet. This translator implementation serves to show that the translation is not only feasible, but is also lightweight enough to be practical, especially for use in such applications as embedded systems.

References

[1] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, S. Eggers, C. Chambers, *Extensibility, Safety and Performance in the SPIN Operating System*, Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Dec. 1995.

[2] M.E. Fiuczynski, V.K. Lam, B.N. Bershad, *The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator*, Proceedings of the 1998 USENIX Technical Conference, New Orleans, LA., Jun. 1998.

[3] M.E. Fiuczynski, B.N. Bershad, *An Extensible Protocol Architecture for Application-Specific Networking*, Proceedings of the 1996 USENIX Technical Conference, San Diego, CA., Jan. 1996.

[4] S. Deering, R. Hinden, *IPv6 Specification*, RFC 1883, Dec. 1995.

[5] A. Conta, S. Deering, *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification,* RFC 1885, Dec. 1995.

[6] E. Nordmark, *Stateless IP/ICMP Translator (SIIT),* IETF Internet Draft, Nov. 1997.

[7] J. Postel, *Internet Control Message Protocol*, RFC 792, Sep. 1981.