The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator

Marc E. Fiuczynski Vincent K. Lam Brian N. Bershad
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

Abstract

IPv6 is a new version of the internetworking protocol designed to address the scalability and service shortcomings of the current standard, IPv4. Unfortunately, IPv4 and IPv6 are not directly compatible, so programs and systems designed to one standard can not communicate with those designed to the other. IPv4 systems, however, are ubiquitous and are not about to go away "over night" as the IPv6 systems are rolled in. Consequently, it is necessary to develop smooth transition mechanisms that enable applications to continue working while the network is being upgraded. In this paper we present the design and implementation of a transparent transition service that translates packet headers as they cross between IPv4 and IPv6 networks. While several such transition mechanisms have been proposed, ours is the first actual implementation. As a result, we are able to demonstrate and measure a working system, and report on the complexities involved in building and deploying such a system.

1 Introduction

The current internetworking protocol, IPv4 [11], eventually will be unable to adequately support additional nodes or the requirements of new applications. IPv6 is a new network protocol that features improved scalability and routing, security, ease-of-configuration, and higher performance compared to IPv4. Unfortunately, IPv6 is *incompatible* with IPv4 and to use the new protocol will require changes to the software in every networked device. IPv4 systems, however, are ubiquitous and are not about to go away "over night" as the IPv6 systems are rolled in. Consequently, it is necessary to develop transition mechanisms that enable applications to continue working while the hosts and networks are being upgraded. One suggested strategy is to translate IP headers as they cross between IPv4 and IPv6 networks [3]. The requirement of header translation is to remain transparent to applications and the network.

In this paper we present two variations of IPv6/IPv4 translators that address these difficulties. The first variation uses *special* IPv6 addresses, as proposed in [4], to easily translate packets transparently for all applications. Unfortunately, these special IPv6 addresses also require IPv6 routers to contain special routes to them, which is considered to be a bad idea because it creates more state for the router to maintain [4]. The second variation maintains an explicit mapping between IPv4 and IPv6 addresses, and is therefore able to use standard IPv6 addresses that do not require any special treatment by IPv6 routers. Its drawback is that IP-addresses embedded in some applications' data stream, such as FTP, must be updated as well for the translation to be completely transparent.

We have built an IPv6/IPv4 network address and protocol translator as a device driver running in the Windows NT operating system [15]. Our test environment consists of the translator as a gateway between IPv6 and IPv4 hosts connected to separate Ethernet segments, and it incurs little performance overhead. Between a pair of IPv6 and IPv4 nodes communicating via the translator, we have measured TCP bandwidth of 7210 Kbytes/second and roundtrip packet latencies of 424 microseconds over 100Mbit/second Ethernet links.

1.1 Motivation

Our efforts began with an implementation of the IPv6 protocol for the SPIN [13] extensible operating system, which enables the rapid prototyping of kernel extensions. After completing the initial IPv6 implementation we connected our system to the 6Bone [12]. We were interested in accessing services using IPv6, but quickly discovered that there were only a few hosts (roughly 250) accessible via the 6Bone with even fewer IPv6 native services to talk to. Thus, we decided to build an IPv6/IPv4 translator to enable IPv6 systems to access the IPv4 systems and services, and vice versa.

There are two main scenarios where network address and protocol translation are applicable:

- An IPv6 site communicating with IPv4 nodes. For example, a completely new network with new devices that all support IPv6 may occasionally need to communicate with some IPv4 nodes out on the Internet.
- An IPv4 site communicating with IPv6 nodes. For example, upgrading an IPv4 site to IPv6 on a node-by-node basis requires that critical services, such as

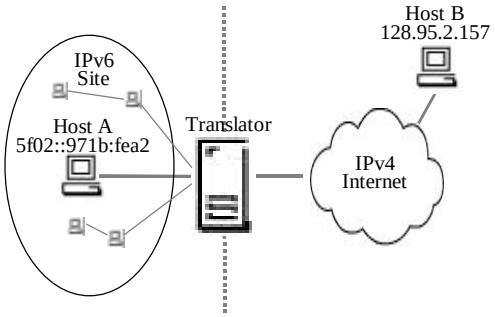

Figure 1. Translator for an IPv6 site.

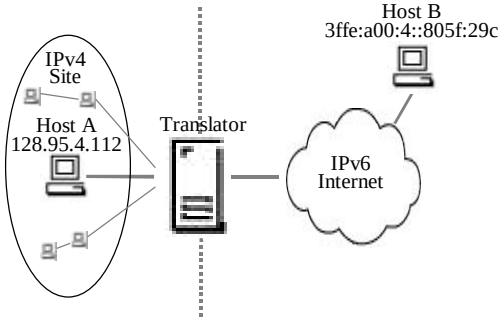

Figure 2. Translator for an IPv4 site.

web, file, and print services are accessible from both IPv6 and IPv4 nodes.

The rest of this paper describes the design and implementation of the IPv6/IPv4 translator and is organized as follows. In Section 2 we describe network address and protocol translation. In Section 3 we present the applications and benchmarks used to test the translator. In Section 4 we discuss possible solutions for some unresolved issues. In Section 5 we survey related work regarding network address and protocol translation. Finally, in Section 6 we conclude.

2 Network Address and Protocol Translation

The address and protocol translation presented in this section enables both the communication between nodes in an IPv4 site with nodes in the IPv6 network, and between nodes in an IPv6 site with nodes in an IPv4 nodes. Figures 1 and 2 illustrate these scenarios, and the following paragraphs describe them in more detail.

Figure 1 illustrates a translator for an IPv6 site communicating with nodes in an IPv4 network. The internal routing of the IPv6 site must be configured such that packets intended for IPv4 nodes route to the translator. Hosts in the IPv6 site send packets to nodes in the IPv4 network using IPv6 addresses that map to individual IPv4 hosts. For this scenario, a design presented in [4] proposes that IPv6 nodes use an *IPv4-compatible IPv6* address as their own address and an *IPv4-mapped IPv6* address when communicating with IPv4-only nodes. An IPv4-compatible IPv6 address holds an IPv4 address in the low-order 32-bits, with a unique high-order 96-bit prefix of `0:0:0:0:0:0` (all zero bits), and always identifies an IPv6/IPv4 or IPv6-only node; they never identify an IPv4-only node. Similarly, an IPv4-mapped IPv6 address identifies an IPv4-only node and its high-order 96-bits bear the prefix `0:0:0:0:0:FFFF`. The address of any IPv4-only node may be mapped into the IPv6 address space by prefixing `0:0:0:0:0:FFFF` to its IPv4 address. The benefit of this approach is that the translator can be stateless. However, regardless of the 96-bit IPv6 prefix that is used to map between the IPv4 and IPv6 address domains it still remains necessary to identify a host in the IPv6 site with an unique IPv4 address. That is, in Figure 1, for Host B to communicate with Host A requires an IPv4 address that can be routed through the IPv4 Internet. To overcome this limitation a stateful translator could multiplex several IPv6 hosts onto a single, globally unique IPv4 address using the TCP/UDP port translation technique described in [2].

Figure 2 illustrates a translator for an IPv4 site communicating with nodes in an IPv6 network. Hosts in the IPv4 site send packets to nodes in the IPv6 network using IPv4 destination addresses assigned by the translator that map to individual IPv6 hosts. For this to work, the internal routing of the IPv4 site must contain routes to the translator for packets with the destination field using one of these IPv4 addresses. The translator, upon receiving such packets, will do the IPv4-to-IPv6 translation and forward the packet to the IPv6 network. In contrast to the above scenario, the translator can use unique IPv6 addresses to refer to nodes in the IPv4 site in order to do IPv6-to-IPv4 translation for packets it receives from the IPv6 network. These IPv6 addresses may come from a pool that is dynamically assigned to the set of IPv4 hosts communicating with IPv6 hosts. A better approach is to assign unique and routable IPv6 addresses to all nodes in the IPv4 site and to register them with DNS. This should be easily possible given that the IPv6 address space is sufficiently large, and also has the benefit that arbitrary hosts in the IPv6 Internet can easily lookup and initiate sessions with nodes in the IPv4 site via the translator.

In summary, the subtle difference between these two scenarios is that the former involves mapping a pool of *global* IPv4 addresses referring to IPv6 addresses, whereas the latter can leverage site *private* IPv4 addresses to refer to IPv6 addresses. Global IPv4 addresses will be scarce and mechanisms are required to dynamically assign a pool of these IPv4 addresses on a temporary basis to IPv6 nodes so that they can

communicate with IPv4 nodes. On the other hand, there is a large pool of roughly 17 million site private IPv4 addresses defined by [14], which can be used by the translator to map to IPv6 addresses. Our translator is designed to support all of the scenarios just described.

To enable communication between an IPv4 and IPv6 node, a translator needs to do both address and protocol translation. Protocol translation involves mapping most of the fields illustrated in Figure 3 from one version of IP to the other. Address translation involves converting addresses for packets crossing the protocol boundary.

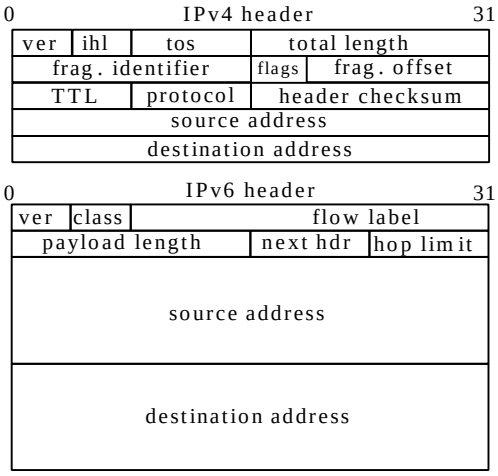

Figure 3. IPv4 and IPv6 header format.

The following two subsections describe the address and protocol translation process in further detail.

2.1 Address Translation

Address translation is trivial when using IPv4-mapped and IPv4-compatible IPv6 addresses. For the IPv6-to-IPv4 direction the translator simply extracts the lower 32-bits of an IPv6 address to obtain an IPv4 address. For the opposite direction the translator sets the lower 32-bits of the IPv6 source/destination addresses to the IPv4 source/destination addresses, and sets the upper 96-bits of the IPv4 source and destination addresses to the IPv4-mapped and IPv4-compatible prefix, respectively. However, it is considered to be a very bad idea to use IPv4-mapped address as it has the drawback of requiring IPv6 routers to contain routes to IPv4-mapped addresses [4]. The alternative is to use IPv6-only addresses to refer to IPv4 nodes, which requires the translator to maintain an explicit mapping between IPv4 and IPv6 addresses.

For clarity, we introduce an IPxNODEy notation to disambiguate among the types of addresses used in the translation process. Table 1 defines the four types of addresses in terms of this notation. The first two rows define the addresses that are native to the IPv4 and IPv6 nodes. The last two rows define address aliases, which

IpxNODEy	Definition
IP4NODE4	v4 address of a 4 node
IP6NODE6	v6 address of a v6 node
IP6NODE4	v6 address referring to a v4 node
IP4NODE6	v4 address referring to a v6 node

Table 1. IP address definition.

are assigned by the translator, used to translate between the IPv4 and IPv6 address domains.

As an example of using this IPxNODEy notation consider the following scenario: an arbitrary IPv6-only host wishes to communicate with our IPv4-only web server via the translator. For an IPv6 host to communicate with our IPv4 web server requires an IPv6 address that is an alias (IP6NODE4) address for the web server's native IPv4 host (IP4NODE4) address. Similarly, for the web server to reply to the IPv6 host requires an IPv4 address that is an alias (IP4NODE6) address for the IPv6 host's native (IP6NODE6) address. That is, the translator maps the IP6NODE4 address to the IP4NODE4 address of the web server, and the IP4NODE6 address to the IP6NODE6 address of the IPv6 host.

The translation of addresses has three phases: address binding, address lookup and translation, and address unbinding, which we describe in the following subsections.

2.1.1 Address Binding

Address binding is the phase where an IPv4 address is associated with an IPv6 address and vice versa. The translator maintains key-to-value tuples, listed in Table 2, to map between IPv4 and IPv6 addresses.

Key-to-Value	Definition
IP6NODE4-to-IP4NODE4	v6 addresses mapped to v4 node addresses
IP4NODE6-to-IP6NODE6	v4 addresses mapped to v6 node addresses

Table 2. Mappings between IPv4 and IPv6 addresses used by translation process.

For addresses that are statically mapped, the binding happens when the translator is initialized. If the translator is configured to use IPv4 mapped/compatible IPv6 addresses then all the bindings are implicitly static as they are defined by these special IPv6 addresses. Other static mappings could be setup between arbitrary IPv4 and IPv6 addresses. For example, the binding of addresses for an IPv4 node to an IPv6 node could be done statically by a network manager when assigning IPv6 addresses to existing nodes in the IPv4 site. That is, IP6NODE4-to-IP4NODE4 are static mappings of IPv6 addresses assigned to IPv4 hosts. Otherwise, the

binding between addresses needs to happen dynamically.

IPv6 addresses are larger than IPv4 addresses and it is not possible to create a one-to-one IP4NODE6-to-IP6NODE6 binding. Consequently, it will be necessary to reuse IP4NODE6 addresses to bind them to other IP6NODE6 addresses. In Section 2.1.3 we discuss this issue in more detail.

2.1.2 Address Lookup and Translation

Once a binding is established it can be used for address lookup and translation. The example in Figure 4 illustrates the translation using the IPxNODEy notation defined earlier. When the IPv4 node sends a packet to

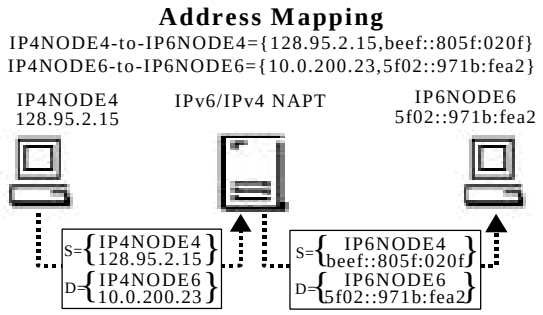

Figure 4. Basic address translation operation.

the IPv6 node it is routed through the translator. The translator receives the packet, translates the 128.95.2.15 to beef::805f:020f source address using the IP4NODE4-to-IP6NODE4 mapping, and translates the 10.95.2.23 to 5f02::971b:fea2 destination address using the IP4NODE6-to-IP6NODE6 mapping. Likewise, IP packets on the return path go through a reverse address translation.

Notice that this requires no changes to hosts or routers. As far as the IPv4 host is concerned, IP4NODE6=10.0.200.23 is the address used by the IPv6 hosts. Conversely, the IPv6 host believes that IP6NODE4=beef::805f:020f is the address used by the IPv4 hosts. The address translation is transparent to both hosts.

2.1.3 Address Unbinding

Address unbinding is the phase when the association between an IPv4 and IPv6 address is broken. We expect the number of bindings of the IP6NODE4-to-IP4NODE4 mapping to remain fairly constant during the day-by-day operation of the translator; new bindings are only necessary when adding new hosts to the site.

On the other hand, the number of bindings of the IP4NODE6-to-IP6NODE6 mapping are more dynamic and depend on the number of connections established to different hosts in the network. The number of reserved IP4NODE6 addresses used by the translator limits the number of bindings possible for the IP4NODE6-to-IP6NODE6 mappings.

For the scenario where the translator is providing service for an IPv6 site (as illustrated in Figure 1), the IP4NODE6 addresses are a small number of unique IPv4 addresses. It is crucial for the translator to detect when an IP4NODE6 address can be reused in order to create new bindings; otherwise, new sessions may be refused if there are no IP4NODE6 addresses available.

For the scenario where a translator is providing service to an IPv4 site (as illustrated in Figure 2), the IP4NODE6 addresses may come from a relatively large pool of private network addresses (as mentioned earlier, there are roughly 17 million of such addresses available). Here the concern is to safely remove unused bindings to ensure that the mapping table does not require too much memory and that address lookup performance does not deteriorate. Removing a binding too early should never occur, as it would effectively terminate any ongoing communication that relied on the binding.

2.2 Protocol Translation

Protocol translation consists of a simple mapping between the two IP protocols, with some special rules for handling fragments and path MTU discovery. The basic operation is to remove the original IP header and replace it with a new header from the other IP version. The rest of this section provides a high-level overview of the protocol translation process and the issues involved. In the Appendix of this paper we present the details of protocol translation between IPv4/IPv6 and ICMPv4/ICMPv6 headers.

2.2.1 IP Translation

The IPv6 and IPv4 headers have some similarity, but there are a number of fields that are either missing or have different sizes or meaning. The translator either directly copies, translates, ignores, or sets fields in the IP header to a default value when translating from one version of IP to the other. Figure 5 illustrates the actions taken by the translator for each header field.

Many of the fields require a simple adjustment. The IPv4 *checksum* field is computed when translating from IPv6-to-IPv4, and ignored when translating from IPv4-to-IPv6. The IPv4 *total-length* field includes the IPv4 header size whereas the IPv6 *payload-length* field does not. The translation needs to account for this difference. The *hop-limit/time-to-live* fields are copied and decreased by one. Finally, the *protocol* field can be directly copied from one version of IP to the other, with ICMPv4 and ICMPv6 protocol numbers being the only exception.

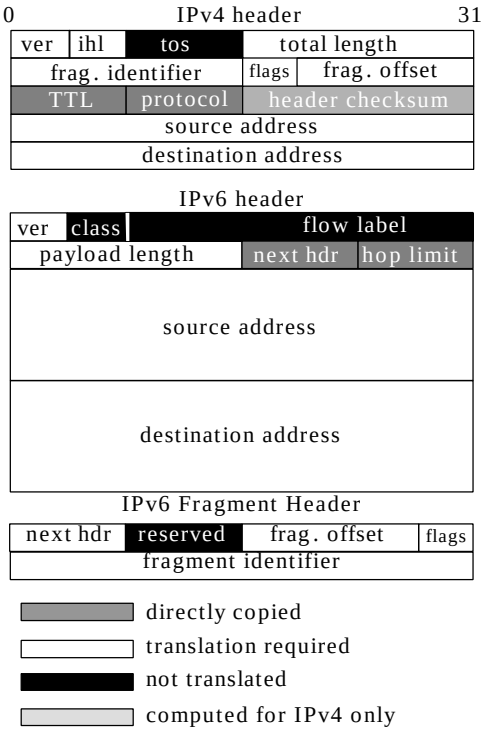

Figure 5. This Figure illustrates which fields of the IPv6/IPv4 header are directly copied, require translation, or are ignored. In contrast to IPv4, the IPv6 header does not have explicit fields to support fragmentation; it uses a separate Fragment header for this information.

With the exception of the IPv6 Fragment header, all other IPv6 extension headers and IPv4 options are silently ignored by the translator. The IPv4 *type-of-service* and IPv6 *traffic-class* and *flow-label* fields are also ignored by the translator, as there does not exist a semantic mapping between them (specifically, the use of the IPv6 flow-label field has not been specified yet). We discuss this loss of information in Section 4.1 further.

When the translator receives a fragmented packet, the translation is straightforward since there is a direct mapping between the IPv4 and IPv6 fragmentation fields. The only caveat is the size difference of the fragment identifier field between the two protocols. In IPv6, this field is 32-bits wide and twice as large as its IPv4 counterpart. To account for this, we currently just copy the lower 16 bits of the IPv6 fragmentation identifier when translating from IPv6 to IPv4.

Whenever the translator encounters a non-fragment IPv4 packet with the *Don't Fragment* flag set to false (i.e., fragmentation is allowed for that packet), it notes that by adding an IPv6 Fragment header and copying the IPv4 fragmentation fields to it, which indicates the following:

1. The sender allows fragmentation and that the fragmentation information is carried end-to-end to ensure that packets are correctly reassembled.
2. The sender is not using path MTU discovery and the *Don't Fragment* bit must be set to false should the packet be translated back to IPv4.

The translation from IPv4 to IPv6 increases the packet size by at least 20 bytes due to the header length difference between the two protocols (28 bytes if it needs to add a Fragment header). If the *Don't Fragment* flag is set to true and the resulting packet is greater than the next-hop MTU, then the translator will return an ICMP error message (Packet Too Big). Otherwise, the translator will fragment the resulting packet into next-hop MTU-sized packets. Note that this fragmentation results in an inefficient packet stream in the case where the IPv4 host is sending MTU-sized packets (e.g., a network file system, such as NFS). For this situation, we are experimenting with returning ICMPv4 "Packet Too Big" error message to the IPv4 host that contains a next-hop MTU that accounts for the size difference in the IP header size, giving the host the opportunity to re-adjust its path MTU value. If the host continues to send large packets (i.e., it does not support path MTU discovery), then the translator will stop sending the ICMP error message and continue fragmenting the packet.

2.2.2 ICMP Translation

The translator silently drops single hop ICMP messages as well as ICMP messages with unknown Type fields. For the remaining ICMP messages the header format is nearly identical for ICMPv4 and ICMPv6. The only exception is the ICMP Parameter Problem message, which an 8-bit pointer value in ICMPv4 and a 32-bit pointer value in ICMPv6. The following ICMP messages and errors have a counterpart in each version: Echo Request, Echo Reply, Time Exceeded, Destination Unreachable, Packet Too Big, and Parameter Problem. For most cases there is a simple translation of the ICMP Type and Code fields. When a Packet Too Big error message reaches the translator, it needs to adjust the Maximum Transmission Unit (MTU) field during the translation to account for the

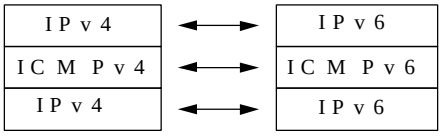

Figure 6. ICMP error messages include the IP header of the error causing packet, which must be translated as well.

difference between IPv4 and IPv6 header sizes. Also, for a Parameter Problem error message the Pointer field needs to be adjusted to point to the corresponding field in the error causing IP header.

ICMP error messages contain as much of the error invoking packet's IP header and data as can fit, and needs to be translated just like a normal IP header that delivered the message. That is, it requires a recursive translation of the IP packet contained in the ICMP error message, as illustrated in Figure 6. The caveat is that the translation of the IP header is likely to change the length of the datagram, in which case the IPv6 Payload-length and IPv4 Total-length fields need to be adjusted as well. Finally, the translator silently drops all IGMP messages.

2.2.3 Adjusting Checksum Values

Several higher-layer protocols (e.g., TCP, UDP) compute their checksum values on a pseudo-header that consists of fields from the IP header. The checksum value needs to be adjusted with the difference between the original IP addresses and the translated IP addresses.

The checksum adjustment for ICMP is slightly more complex. ICMPv6 uses a pseudo-header checksum similar to UDP and TCP, whereas ICMPv4 does not. For ICMP Echo and Echo Request informational messages we calculate the incremental checksum adjustment, as only the Type value changes. When translating from ICMPv6 to ICMPv4 we need to subtract the pseudo-header checksum. Conversely, when translation from ICMPv4 to ICMPv6 we need to add the pseudo-header checksum. Note that these informational messages may be fragmented either by the sending host or intermediate routers if their size exceeds the path MTU. For this case, the translator cannot calculate the correct checksum value for ICMP Echo and Echo Request messages, because it does not know the total size of the packet, which it requires to add/subtract the pseudo-header checksum value when translating between the ICMP versions. Finally, since ICMP error messages are never fragmented, our approach is to recalculate the checksum value from scratch rather than incrementally, because most of the ICMP header and data values have changed.

3 Implementation

In this section we present basic performance measurements and describe a set of applications that we have used to verify whether the translator works for real applications. Our experimental setup consists of IPv6 and IPv4 machines connected to separate, private Ethernet segments. The translator is equipped with two Ethernet cards and acts as a gateway between the IPv6 and IPv4 Ethernet segments. All machines in our setup are Intel PCs equipped with a 200Mhz Pentium Pro processor, 64MB of RAM, and 3COM 3c905 fast Ethernet cards. We use both Linux (2.1.95) and Windows NT 4.0 as our IPv6 test machines. For Windows NT we use Microsoft Research's publicly released IPv6 stack [16]. The translator is implemented as a Windows NT device driver and roughly consists of 2000 lines of C code. It uses the IPv4 and IPv6 stacks in Windows NT to send IP packets.

3.1 Latency and Bandwidth

To evaluate the performance of the translator we used the `ttcp` tool to measure bandwidth and `ping` to measure latency between a pair of IPv6 and IPv4 hosts. We compare the packet forwarding performance of the IPv6/IPv4 translator with NT's built-in IPv4 forwarding support.

We measured the roundtrip latency of `ping` packets ranging in size from 64 bytes to 1440 bytes on 100Mbps Ethernet links. In Table 3, the columns labeled *v4-v4* and *v6-v6* show the latency between two machines communicating directly using the same protocol. The columns labeled *FWD* and *NAPT* show the roundtrip latency going through NT's forwarder and our translator, respectively. The translator is on average about 30 microseconds slower compared to the forwarder.

Msg. size in bytes	v4-v4	v6-v6	FWD	NAPT
64	246	244	397	424
128	262	261	448	463
256	297	295	508	540
512	364	360	630	658
1024	487	482	871	918
1440	603	596	1059	1104

Table 3. Roundtrip latency of PING packets measured in microseconds.

Table 4 shows the bandwidth of sending 64 Mbytes using TCP for both 10Mbps and 100Mbps Ethernet. Note that for 10Mbps Ethernet the overhead of the translator and the forwarder are essentially unnoticeable. However, the bandwidth for the forwarder and the translator on fast Ethernet is much lower compared to two machines communicating directly using either IPv4 or IPv6. Using NT's performance monitor we noticed that processor utilization reaches nearly 100% on our forwarder/translator machine when running the ttcp

Link Speed	v4-v4	v6-v6	FWD	NAPT
Ethernet	1095	1092	1093	1089
Fast Ether	11003	9076	8005	7210

Table 4. TCP bandwidth measured in Kbytes/second.

bandwidth benchmark over fast Ethernet. The reason for the high CPU utilization is NT's packet receive architecture, which assumes the device driver owns the packet buffer rather than passing buffer ownership to the module receiving the packet (as is the case in most UNIX systems). Consequently, we believe that bandwidth through the translator and the forwarder are CPU limited, as they incur significant overhead due to NT's packet receive architecture; they must allocate buffer space for the IP packet's payload and copy the data in its entirety before being able to forward it. Additionally, note that the bandwidth through the translator is 10% slower compared to the forwarder. We attribute this performance degradation partly to the IPv6 prototype from Microsoft Research, which is roughly 1.9Mbytes/second slower than the production IPv4 stack shipped with Windows NT. We expect the end-to-end TCP bandwidth to improve as the IPv6 implementation for Windows NT matures.

We are pleased with the current latency and bandwidth measurements, as they indicate that translation does not inherently have a significant impact on performance.

3.2 Applications

The goal of the translator is to transparently work for "real world" applications, and we used a representative set of programs that exercise the TCP, UDP, and ICMP protocols via the translator. Our test applications consist of an IPv6 version of an Apache web-server, `ttcp`, `finger`, `telnet`, `ping`, `traceroute`, and `ftp`.

We knew from our experiments with `ttcp` that the TCP protocol translation works, but wanted to verify this with common TCP applications. We were able to use `telnet` and `finger` to connect between IPv6 and IPv4 hosts through the translator. Additionally, a web browser on an IPv4 host retrieving documents from an IPv6 Apache web-server was equally successful.

The `ping` program uses ICMP messages to determine whether a particular host is alive. We also used `ping` to measure basic roundtrip latency between hosts.

The `traceroute` program tracks the flow of a packet from router to router. When tracking routes from an IPv6 node through the translator along an IPv4 network, the addresses of the IPv4 routers are translated into IPv4-mapped IPv6 addresses. For the other direction, the translator establishes bindings, described in Section 2.1.1, for the IPv6 router addresses to private network addresses.

Although `ping` and `traceroute` use ICMP, they do not adequately test whether the recursive ICMP translation, described in Section 2.2.2, was working properly. Table 5 lists how we caused various ICMP error messages to verify their correct translation.

Finally, we tested `ftp`, which is an application that embeds an ASCII IP address and sends it to its peer. For it to work correctly via the translator, the IPv6 implementation of the ftp client needs to detect whether the connection is with an IPv6 or IPv4 version of the ftp daemon. When communicating with an IPv4 ftp daemon it needs to use as an ASCII IP address of its host's IPv4-compatible IPv6 address instead of the host's native IPv6 address. Conversely, when an IPv4 ftp client contacts an IPv6 ftp daemon, the daemon must treat the ASCII IP address as an IPv4-mapped IPv6 address. With this approach it is not necessary for the translator to update the ASCII IP address.

ICMP Error Message	Error causing action
Destination unreachable	UDP packet to unreachable port
Packet Too Big	packet exceeding path MTU size
Time Exceeded	single incomplete IP fragment
Parameter Problem	packet with invalid field

Table 5. Error causing actions to verify ICMP translation.

4 Discussion

The previous section illustrated that the basic translation between the two IP protocols is possible for real applications. In this section, we discuss some unresolved issues regarding loss of information, applications with IP address content, and how IPv6 hosts resolve to IPv6 addresses referring to IPv4 hosts (i.e., IP6NODE4 addresses) and vice versa (i.e., IP4NODE6 addresses). Finally, we discuss an integrated translator approach that addresses the host lookup problem and address-unbinding problem mentioned in Section 2.1.3.

4.1 Loss of Information

Although a basic mapping exists between the two IP protocols there are certain fields, options, and extensions that cannot be translated. The result is a loss of information that may have some impact on applications. For example, IPv4 *type-of-service* values cannot be equivalently expressed in an IPv6 context where quality of service for a packet is marked by two fields, *traffic-class* and *flow-label*, as they differ in their currently specified semantics. 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. Further, the IPv6 specification defines extensions for features such as Authentication, Encapsulation, and Extended Routing 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. Our current approach is to ignore all of these features during the translation process and observe the impact on applications. So far our experience is that applications generally rely on basic IP features and do not use the extended fields of the IP header.

4.2 Applications with IP Address Content

Some applications embed their IP addresses in the packet payload, above Layer 3. This is the case for a number of applications, including certain File Transfer Protocol (FTP) programs, and the Windows Internet Name Service (WINS) registration process of Windows 95 and Windows NT. Unless the translator parses every packet all the way to the application level, it has no way of translating embedded IP addresses, which can lead to application failures. Our implementation does not do any application-level IP address translation, but as described in Section 3.2, this is not an issue with new IPv6 applications that are IPv4-aware, like FTP. We hope that a similar solution can be used with IPv6 versions of all legacy applications that embed IP address content. If that's not possible, then the translator will need to be complemented with application level gateways to expand the list of supported applications [7].

4.3 Hostname Lookup

Before a host can initiate a session with another host it has to lookup its address. This is generally done using host tables or DNS. The problem when using a translator is that the lookup needs to resolve to an address alias that refers to the actual host. For the case where the translator enables nodes in an IPv4 site to communicate with nodes in the IPv6 network it is reasonable to assume that each IPv4 node has assigned to it a unique IPv6 address. Thus, arbitrary IPv6 nodes can lookup its address and initiate a session. However, the converse of an IPv4 looking up an IPv6 host is more difficult, as the IPv4 node needs to obtain the address alias from the translator that refers to the IPv6.

There are several approaches that can be taken to translate an IPv6 DNS record to an IPv4 DNS record. First, the resolver library of the IPv4 nodes could be modified to request the alias from the translator when encountering IPv6 DNS records. Second, the site's DNS servers could be modified to request a temporary address from the translator on behalf of its IPv4 clients when encountering an IPv6 DNS record. Finally, an approach proposed in [7] suggests that the translator recognize DNS request and response packets and translates them transparently.

The implications that IPv6/IPv4 translation has to DNS are beyond the scope of this paper, but need to be addressed for translation to be completely transparent.

4.4 The Integrated Approach

Our experience with a network-based translator revealed that for IPv6/IPv4 translation to be completely transparent requires varying degrees of integration with other services. As mentioned in the previous subsection, some level of cooperation is necessary between DNS and the translator to bind IPv4 addresses to IPv6 addresses, and vice versa. Also, the translator currently uses ad-hoc methods to detect when it can safely remove bindings. Our strategy is to integrate the translator functionality directly into an IPv6/IPv4 host operating system. There are several benefits of the integrated approach:

- *Failure isolation*. The integrated translator only serves the host that it is running on and its failure will not affect other hosts.
- *Scalability*. The integrated translator needs to scale only with the number of network applications running on the host, rather than the sum of network applications running in the site served by a network based translator.
- *Safe reclamation of address bindings*. The integrated translator is aware when an application terminates a TCP/UDP network connection and can safely unbind the address.

Finally, and most noteworthy, the integrated approach enables the illusion of an IPv6-only node, as packets stemming from legacy IPv4 applications may be translated to IPv6 before they leave the machine.

5 Related Work

In principle the function of IPv6/IPv4 address translation is similar to an IPv4 Network Address Translator (NAT) [2], which converts private internal addresses to globally unique addresses that are passed to the Internet backbone and vice versa. The IPv4 NAT has the following limitations. First, it is stateful in order to map between the globally unique and private internal addresses; thus the NAT is a single point of failure. Second, applications with IP-address content require special translation that may be difficult (such as updating ASCII IP strings and maintaining TCP sequence numbers on the fly), or may be impossible when the application data stream is encrypted or signed. Any stateful translator shares these limitations. Nevertheless, despite these limitations NATs seem to be widely used.

A proposal called "Network Address Translation – Protocol Translation" (NAT-PT) [7] presents a stateful IPv6/IPv4 translator design. It also describes how to incorporate IPv4 NAT style UDP/TCP port number translation. With exception of the port number translation this is similar to the stateful component of our design.

A proposal called “Stateless IP/ICMP Translation" (SIIT) [4] avoids the need for address translation, thereby overcoming the limitations of IPv4 NAT. First, it does not maintain state, and is therefore resilient to network failure. Moreover, multiple stateless translators may be used to scale with larger sites. Second, the use of IPv4-mapped and IPv4-compatible addresses allows it to avoid translating IP addresses embedded in the application’s data stream. However, this approach will only work if the IPv6 socket API treats mapped/compatible addresses exactly as IPv4 addresses. For example, as is the case for some FTP programs, mapped/compatible IPv6 addresses need to be printed as IPv4 ASCII strings. The drawback of the SIIT design is that IPv6 routers need to contain routes to IPv4-mapped addresses. This drawback seems acceptable when the translator serves an IPv6 site with access to the IPv4 Internet (e.g., the scenario shown in Figure 1). However, for the case where the translator serves an IPv4 site with access to the IPv6 Internet (e.g., the scenario shown in Figure 2) the use of IPv4-mapped/compatible IPv6 address is unreasonable, as it counteracts one of the significant benefits of IPv6: shrinking backbone routing tables.

Finally, a proposal called “Assignment of IPv4 Global Addresses to IPv6 Hosts” (AIIH) [5] enables dual-stack IPv6/IPv4 nodes to temporarily acquire a global IPv4 address to communicate with other IPv4-only nodes. This approach may be the initial stepping stone to allow sites to configure a large set of IPv6 hosts without having to statically assign each host a globally unique IPv4 address.

Both the SIIT and AIIH designs focus on providing interoperability between an IPv6 site and the IPv4 Internet, whereas stateful translation (e.g., NAT-PT) enables an IPv4 site to communicate with the emerging IPv6 Internet.

While several translator designs have been proposed [4][7], ours is the first actual implementation. Our translator implementation is based on the address translation techniques described in Section 2.1, which are general enough to support both stateful and stateless translation.

6 Conclusion

We have described the design and implementation of an IPv6/IPv4 network address and protocol translator, and briefly compared pros and cons of stateless vs. stateful translation. To this date there are three proposals [4][5][7] submitted to the IETF NGTRANS working group to support the interoperability between IPv6 and IPv4-only nodes. Our work subsumes both the stateless SIIT design described in [4] and the stateful design described in [7]. Despite the limitations of translation (e.g., loss of information) we believe that a translator can adequately fulfill the role of a short-term transition aid from IPv4 to IPv6, since it supports the majority of Internet traffic (HTTP, FTP, sendmail).

Based on our experience we conclude that an IPv6/IPv4 network address and protocol translator is complementary to the AIIH [5] approach in transitioning from IPv4 to IPv6. In particular, we believe that it will be a valuable tool to developers porting applications from IPv4 to IPv6. For instance, a server application ported to IPv6 can be tested without having to port the client as well.

For more information about the IPv6/IPv4 translator, its performance, and source availability, please visit our web page at:
www.cs.washington.edu/research/networking/napt

References

[1] S. Deering and R. Hinden. Internet Protocol, Version 6. RFC 1883, December 1995.

[2] P. Srisuresh and K. Egevang. The IP Network Address Translator (NAT). RFC 1631, May 1994.

[3] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 1933, April 1996.

[4] E. Nordmark. Stateless IP/ICMP Translator (SIIT). Work In Progress.

[5] J. Bound. Assignment of IPv4 Global Addresses to IPv6 Hosts (AIIH). Work In Progress.

[6] R. E. Gilligan, S. Thomson, J. Bound, and W. R. Stevens. Basic Socket Interface Extensions for IPv6. Work In Progress.

[7] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF Internet Draft, March 1998. Work In Progress.

[8] J. Mogul and S. Deering. Path MTU Discovery, RFC 1191, November 1990.

[9] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6, RFC 1981, Aug. 1996.

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

[11] J. Postel. Internet Protocol. RFC 791, Sept. 1981.

[12] B. Fink, 6Bone Overview and Links. http://www.6bone.net

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

[14] Y. Rekhter, B. Moskowitz, D. Karrenberg, and G. de Groot. Address Allocation for Private Internets. RFC 1597, March 1994.

[15] H. Custer. Inside Windows NT. Microsoft Press. 1993.

[16] R. P. Draves, A. Mankin, and B. D. Zill. Implementing IPv6 for Windows NT. Proceedings of the 2nd USENIX NT Symposium, Aug. 1998.

A. Protocol Translation Details

This appendix describes the protocol translation for both IP and ICMP headers in detail. The translation description is based on the text from [4] with minor corrections based on our implementation experience.

A.1 Translating IPv4 to IPv6 Headers

If the Don't Fragment flag is true and the IPv4 packet is not a fragment (i.e., the More Fragments flag is false and the Fragment Offset is zero) then the IPv6 header fields are set as follows:

- Version: 6
- Traffic-Class: 0 (all zero bits)
- Flow ID: 0 (all zero bits)
- Payload Length: Total Length value from IPv4 header, minus the Internet Header Length (multiplied by 4) value from the IPv4 header
- Next Header: Protocol field copied from IPv4 header. If the value of the Protocol field is 1 (ICMPv4), then substitute it with 58 (ICMPv6)
- Hop Limit: Time To Live value from IPv4 header decreased by one
- Source and Destination Addresses: Depends on address translation mechanism

If there is need to add a Fragment header (i.e., the Don't Fragment flag is false or the More Fragments flag is true or the Fragment Offset is non-zero) the IPv6 header fields are set as above with the following exceptions:

- Payload Length: Total Length minus the Internet Header Length (multiplied by 4) from the IPv4 header, plus 8 for the Fragment header
- Next Header: 44 (Fragment Header)

The Fragment header fields are set as follows:

- Next Header: Protocol field copied from IPv4 header. If the value of the Protocol field is 1 (ICMPv4), then substitute it with 58 (ICMPv6).
- Reserved: 0 (all zero bits)
- Fragment Offset: Fragment Offset copied from the IPv4 header.
- M flag: More Fragments flag copied from the IPv4 header.
- Identification: The low-order 16 bits copied from the Identification field in the IPv4 header. The high-order 16 bits set to zero.

A.2 Translating IPv6 to IPv4 Headers

With exception of the IPv6 Fragment header, all other IPv6 extension headers are ignored (i.e., there is no attempt made to translate them). For each IPv6 extension header that is ignored the Payload Length needs to be adjusted by the size of these headers before the IPv4 Total Length field is calculated.

If there is no IPv6 Fragment header the IPv4 header fields are set as follows:

- Version: 4
- Internet Header Length: 5 (no IPv4 options)
- Type of Service: 0 (all zero bits)
- Total Length: Payload length value from IPv6 header, plus the size of the IPv4 header.
- Identification: 0 (all zero bits)
- Flags: Don't Fragment flag is set to true (1), and all other flags set to false (0)
- Fragment Offset: 0 (all zero bits)
- Time To Live: Hop Limit value from IPv6 header decreased by one
- Protocol: Next Header copied from IPv6 header or last extension header; and, if the value of the Next Header field is 58 (ICMPv6), then substitute it with 1 (ICMPv4)
- Header Checksum: Computed once the IPv4 header has been created
- Source and Destination Address: Depends on address translation mechanism

If the IPv6 packet contains a Fragment header the header fields are set as above with the following exceptions:

- Total Length: Payload length value from IPv6 header, minus 8 for the Fragment header, plus the size of the IPv4 header.
- Identification: Copied from the low-order 16-bits in the Identification field in the Fragment header.
- Flags: The More Fragments flag is copied from the Fragment header and the Don't Fragments flag is set to false.
- Fragment Offset: Copied from the Fragment Offset field in the Fragment Header.

A.3 Translating ICMPv4 to ICMPv6

Echo and Echo Reply (Type 8 and Type 0): set the Type to 128 and 129, respectively.

Destination Unreachable (Type 3): for most Code values set the Type to 1, unless specified otherwise below. Translate the Code field as follows:

- Code 0, 1, 6, 7, 8, 11, and 12: set Code to 0 (no route to destination)
- Code 2: translate to an ICMPv6 Parameter Problem (Type 4, Code 1) and set the Pointer to 6, which is the IPv6 Next Header field
- Code 3: set Code to 4 (port unreachable)
- Code 4: translate to an ICMPv6 Packet Too Big message (Type 2, Code 0) and the MTU field needs to be adjusted for the difference between the IPv4 and IPv6 header sizes

- Code 5: set Code to 2 (not a neighbor)
- Code 9, 10: set Code to 1 (communication with destination administratively prohibited)
- Time Exceeded (Type 11): set the Type field to 3. The Code field is unchanged
- Parameter Problem (Type 12): set the Type field to 4 and translate the Pointer values as follows: 0-to-0, 2-to-4, 8-to-7, 9-to-6, 12-to-8, 16-to-24, and for all other ICMPv4 Pointer values set the ICMPv6 Pointer value to –1.

A.4 Translating ICMPv6 to ICMPv4

Echo Request and Echo Reply (Type 128 and 129): set the Type to 0 and 8, respectively.

Destination Unreachable (Type 1): set the Type field to 3. Translate the code field as follows:

- Code 0: Set Code to 1 (host unreachable)
- Code 1: set Code to 10 (communication with destination host administratively prohibited)
- Code 2: set Code to 5 (source route failed)
- Code 3: set Code to 1 (host unreachable)
- Code 4: set Code to 3 (port unreachable)

Packet Too Big (Type 2): translate to an ICMPv4 Destination Unreachable with code 4. The MTU field needs to be adjusted for the difference between the IPv4 and IPv6 header sizes taking into account whether or not the packet in error includes a Fragment header

Time Exceeded (Type 3): set the Type to 11. The Code field is unchanged.

Parameter Problem (Type 4): If the Code is 2 then set Type to 12, Code to 0, and Pointer to –1. If the Code is 1 translate this to an ICMPv4 protocol unreachable (Type 3, Code 2) message. If the Code is 0 then set the Type to 12, the Code to 0, and translate the Pointer values as follows: 0-to-0, 4-to-2, 7-to-8, 6-to-9, 8-to-12, 24-to-16, and for all other ICMPv6 Pointer values set the ICMPv4 Pointer value to –1.

Author Information

Marc E. Fiuczynski (mef@cs.washington.edu) is a Ph.D. student in Computer Science and Engineering at the University of Washington. His research interests are internetworking, operating systems, extensible systems, and intelligent I/O systems. He received his B.A. in Computer Science from Rutgers College in 1992 and his M.S. in Computer Science and Engineering from the University of Washington in 1995. He's worked for several years on the SPIN extensible operating system and hopes to complete his Ph.D. degree before the next millennium.

Vincent K. Lam (vkl@cs.washington.edu) is an undergraduate student in Computer Science and Engineering at the University of Washington, and graduates in June 1998 with a B.S. degree.

Brian N. Bershad (bershad@cs.washington.edu) is an Associate Professor in Computer Science and Engineering at the University of Washington. His research interests include operating systems, distributed systems, networking, parallel systems, and architecture.